VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Synchronizace datových úložišť

Data Storage Synchronizations

2018 Ludvík Hobza

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Ludvík Hobza**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Synchronizace datových úložišť
Data Storage Synchronizations

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je prozkoumat jaký je současný stav v přístupu k synchronizaci oddělených datových úložišť, které využívají různé aplikace. Při synchronizaci se nejedná o replikaci celého úložiště, pouze se získávají data, která jsou relevantní z hlediska přístupu pro uživatele, který používá danou aplikaci. Typicky se jedná získání dat ze serverové části aplikace do mobilního zařízení za účelem umožnění „offline" provozu. Dalším krokem je návrh vlastního přístupu a jeho implementace jako knihovny, aplikačního rámce či zásuvného modulu.

Postup pro vypracování:
1. Proveďte průzkum současného stavu.
2. Navrhněte vhodný přístup.
3. Vytvořte implementaci.
4. Proveďte testování. Otestujte také výkonnostní parametry daného řešení.

Práce bude zejména obsahovat:
1. Popis současného stavu.
2. Návrh vlastní knihovny či "frameworku", který obecně řeší požadovanou funkčnost.
3. Prototypovou implementaci návrhu z bodu 2., která bude dostupná v repozitáři aplikace běžící na git.cs.vsb.cz.
4. Skripty pro automatizované testování funkčnosti a výkonosti vytvořeného software.

Seznam doporučené odborné literatury:

[1] TANENBAUM, Andrew S. a Maarten van STEEN, 2016. Distributed Systems: Principles and Paradigms. 2 edition. Leiden: CreateSpace Independent Publishing Platform. ISBN 978-1-5302-8175-6.
[2] BESTA, Maciej, Michal PODSTAWSKI, Linus GRONER, Edgar SOLOMONIK a Torsten HOEFLER, 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing [online]. New York, NY, USA: ACM, s. 93–104 [vid. 2017-10-10]. HPDC '17. ISBN 978-1-4503-4699-3. Dostupné z: doi:10.1145/3078597.3078616

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2018 .....

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce Ing. Janu Kožusznikovi, Ph.D. za pomoc a připomínky k vypracování této práce.

Abstrakt

Tato diplomová práce se zabývá synchronizací dat v oddělených datových úložištích. Nejedná se však o úplnou synchronizaci, kde se synchronizují všechna data mezi úložišti, ale o synchronizaci dat, která jsou relevantní pro uživatele využívajícího danou aplikaci. V práci je nejdříve proveden průzkum současného stavu tohoto přístupu a poté je navrhnuta knihovna, která tuto problematiku řeší. Následně je implementován prototyp navržené knihovny společně s ukázkovým systému, který tuto knihovnu využívá. Nakonec je implementované řešení otestováno.

Klíčová slova: Java, databáze, datové úložiště, synchronizace, distribuovaný systém, offline provoz

Abstract

This diploma thesis deals with the synchronization of data in separated data storages. However, this is not a full synchronization where all the data between the data storages are synchronized, but a data synchronization where only data that are relevant to the users of the application are synchronized. The paper first examines the current state of this approach and then designs a library that addresses this issue.The prototype of the proposed library is then implemented along with the sample system that uses this library. Finally, the implemented solution is tested.

Key Words: Java, database, data storage, synchronization, distributed system, offline traffic

Obsah

Seznam použitých zkratek a symbolů

ADB	–	Android Debug Bridge
API	–	Application Programming Interface
CRUD	–	Create Read Update Delete
DAO	–	Data Access Object
DOM	–	Document Object Model
GUID	–	Globally Unique Identifier
HTTP	–	Hyper Text Transfer Protocol
HTTPS	–	HTTP Secure
IETF	–	Internet Engineering Task Force
JSON	–	JavaScript Object Notation
JPA	–	Java Persistence API
JPEG	–	Joint Photographic Experts Group
MD	–	Message-Digest
MVC	–	Model View Controller
ORM	–	Objektově Relační Mapování
PDF	–	Portable Document Format
POM	–	Project Object Model
PNG	–	Portable Network Graphics
REST	–	Representational State Transfer
SAX	–	Simple API for XML
SHA	–	Secure Hash Algorithm
SSL	–	Secure Sockets Layer
SVG	–	Scalable Vector Graphics
SQL	–	Structured Query Language
TLS	–	Transport Layer Security
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
USB	–	Universal Serial Bus
UUID	–	Universally Unique Identifier
XML	–	Extensible Markup Language
WAR	–	Web application ARchive

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době téměř každý vlastní chytrý mobilní telefon a/nebo jiné podobné zařízení. Spoustu aplikací používaných na těchto zařízeních při svém běhu synchronizují svá data se serverem. Pokud jsou tato data upravena, aplikace si stáhne jejich nejnovější verzi a uživatel tak vždy pracuje s nejaktuálnější verzí těchto dat. Jestliže uživatel data upraví, jsou tyto úpravy odeslány na server, odkud si jej mohou stáhnout ostatní uživatelé. Mobilní zařízení však nemusí mít vždy přístup k internetu a může se tedy stát, že aplikace nebude moci vždy svá data se serverem synchronizovat. Uživatel by ale měl mít možnost upravovat svá data i tehdy, když nemá zrovna přístup k internetu, a tyto změny by měly být na server poslány ve chvíli, kdy bude zařízení znovu připojeno k internetu. Může ovšem nastat situace, kdy dva nebo více uživatelů upraví stejná data ve chvíli, kdy jsou odpojeni od internetu, a při synchronizaci těchto dat dojde ke konfliktu jejich úprav. Při synchronizaci dat mobilního zařízení by navíc měla být posílána pouze data, která jsou pro daného uživatele relevantní, aby zbytečně nebyla zatěžována síť a aby paměť mobilního zařízení nebyla zbytečně zaplněna.

Cílem diplomové práce je prozkoumat současný stav tohoto přístupu a následně navrhnout a vytvořit prototypovou implementaci knihovny, která by zmíněný přístup realizovala. Průzkum současného stavu dané problematiky je popsán v první kapitole této práce. Jsou zde uvedeny zdroje, ze kterých se při průzkumu čerpaly informace, a dosažené výsledky tohoto průzkumu. Návrh knihovny a vzorové implementace systému, ve kterém je problematika řešena, je popsán v druhé kapitole. V této kapitole je také popsáno řešení problémů, které se během návrhu vyskytly. Poslední kapitola je zaměřena na testování navrženého systému a knihovny. Popisuje metody a nástroje použité při funkčním testování knihovny a výkonnostním testování vzorového systému, u kterého probíhalo i testování rozhraní. Nakonec jsou prezentovány výsledky výkonnostního testování.

2 Průzkum současného stavu

Před návrhem knihovny, který je popsán v následující kapitole, byl proveden průzkum, jestli už neexistuje nějaké řešení dané problematiky. Prvním krokem průzkumu bylo prohledání akademických portálů *Scopus* [1] a *Web of Science* [2], kde po prozkoumání článků zabývajících se sdílenou pamětí, částečnou synchronizací a datovými úložišti, bohužel nebyly nalezeny žádné informace o odborné literatuře, která by se přesně zabývala problematikou, kterou řeší tato práce.

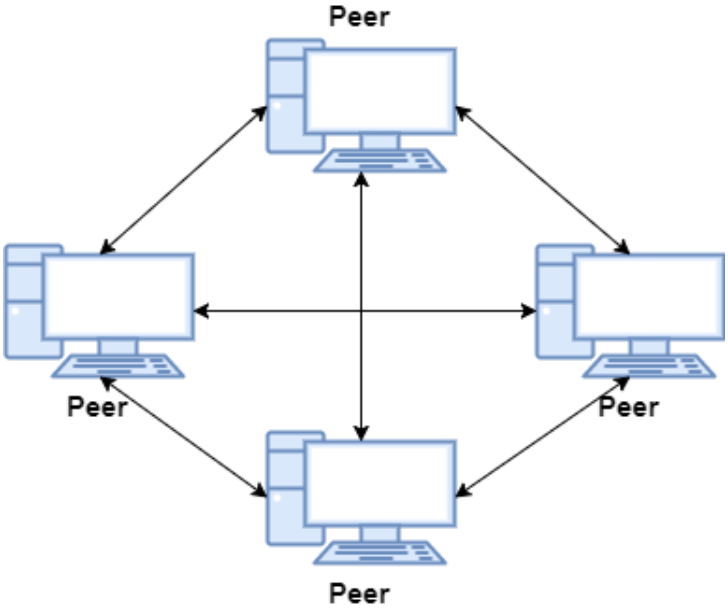
Dalším krokem proto bylo prohledávání také v klasických vyhledávačích, jako např. Google, a byl nalezen pouze jeden relevantní článek z roku 2006, který se tímto tématem částečně zabývá, avšak ten řeší pouze částečnou synchronizaci, nikoliv však synchronizaci bez možného odpojení od sítě. Jedná se o článek nazvaný *Optimistic Algorithms for Partial Database Replication* [3]. Tento článek je stručně popsán v další podkapitole. Po důkladném pročtení článku byl zaslán dotaz jednomu z autorů, zda od doby, kdy byl článek vydán, nedošlo k nějakému dalšímu vývoji v této oblasti. Ten však na tento dotaz neodpověděl.

V knize *Distributed Systems: Principles and Paradigms*[6] se věnuje problematice replikace kapitola nazvaná *Consistency and replication*. V této kapitole jsou popsány modely konzistence dat distribuovaných systémů, a také způsob, jak tyto modely implementovat. Bohužel žádný z těchto modelů nelze aplikovat na takový systém, jehož klientská část nemusí mít vždy přístup k serverové části tohoto systému. Byla také prostudována kniha *Patterns of Enterprise Application Architecture*[7], ve které se jedna její část věnuje problematice souběhu a distribuovaných systémů. Jsou zde uvedeny i návrhové vzory, které pomáhají tyto problémy řešit. Stejně jako u předchozí knihy neberou popsané přístupy v úvahu systémy, u kterých klientská část nemusí mít vždy přístup k serverové části. Jeden návrhový vzor popsaný v této knize byl použit jako inspirace pro návrh klientské části navrhované knihovny. Vedoucí práce Ing. Jan Kožusznik, Ph.D., který se touto problematikou v minulosti zabýval, pro tuto práci poskytnul svůj článek nazvaný *Mobile Device Synchronization with Central Database based on Data Relevance*[40].

Byl také proveden průzkum softwaru, jenž by tento problém řešil, ať už by se jednalo o framework, knihovnu nebo hotový produkt. Dalo by se předpokládat, že by mohl existovat software, který tento problém řeší alespoň z části, protože je velmi pravděpodobné, že existuje spousta aplikací, které se během svého vývoje touto problematikou zabývaly. Jedná se např. o různé hry, chatovací aplikace atd. Byly však nalezeny pouze frameworky, které se zabývají úplnou synchronizací dat, jako například *Couchbase mobile* [4]. Posledním krokem průzkumu bylo zjistit, jak se tato problematika řeší v praxi. Byl kontaktován vývojář z firmy Kvados a. s., který poskytl informaci, že na tuto problematiku u svých projektů narazili, nepoužili však žádné obecné řešení, ale postupovali ad-hoc.

2.1 Optimistic Algorithms for Partial Database Replication

Tento článek popisuje dva algoritmy pro částečnou replikaci databáze, kde je systém rozdělen na jednotlivé databázové části, které se zde nazývají stránky. Tyto stránky mají částečnou kopii databáze, komunikují vzájemným zasíláním zpráv a nemají žádnou sdílenou paměť. Jedná se tedy o peer-to-peer architekturu (viz Obrázek 1), kde spolu jednotliví klienti komunikují přímo, asynchronně a bez jakéhokoliv centrálního prvku.



Obrázek 1: Architektura Peer-to-Peer

Pro operace čtení a zápisu dat se používají transakce. Transakce se může nacházet v jednom z těchto stavů:

- **Prováděna** – Transakce pro zápis a čtení jsou prováděny lokálně na základě dvoufázového zamykání. Pokud u transakce dochází pouze ke čtení, je automaticky potvrzena a přejde do potvrzeného stavu. Pokud transakce provádí zápis, tak je nejdříve poslána na validaci a poté přejde do odeslaného stavu.
- **Odeslána** – Když transakce přejde do odeslaného stavu, všechny její zámky jsou uvolněny a transakce se stane eventuálně validní. Validace zajistí, že když jiná potvrzená transakce T' bude vykonána ve stejnou dobu jako transakce T a T bude číst data zapsaná transakcí T', tak bude transakce T zrušena.
- **Potvrzena** – Transakce byla vykonána – finální stav.
- **Zrušena** – Transakce byla zrušena – finální stav.

Hlasování o validitě

Při přechodu transakce ze stavu „prováděna“ do stavu „odeslána“ musí být nejdříve validována. Stránka se může bezpečně rozhodnout, jestli transakci zahodí nebo potvrdí pouze tehdy, pokud obdržela všechny hlasy z hlasovací skupiny. Tato hlasovací skupina pro danou transakci je tvořena všemi stránkami, které obsahují data, která tato transakce mění. Aby byla transakce potvrzena, musí všechny stránky v hlasovací skupině hlasovat ano. Pokud jakákoliv stránka v hlasovací skupině hlasuje ne, znamená to, že transakce má stará data a musí být zrušena. Pokud je transakce schválena, data transakce se zapíší a transakce přejde do potvrzeného stavu.

2.1.1 Popisované algoritmy

Algoritmus “One-at-a-time“

Stránky provádí sekvenci kroků. V každém kroku stránky rozhodnu o výsledku jedné z transakcí. Krok je složen ze dvou fází. První fáze zajistí, že se stránky shodnou na pořadí provádění transakcí. Ve druhé fázi si stránky vymění výsledky jejich validace. Algoritmus se skládá ze tří částí, které jsou vykonávány souběžně:

1. Pokud není fronta nerozhodnutých transakcí prázdná, tak stránka:

   Vybere první transakci z fronty nerozhodnutých transakcí, dá této transakci hlas a navrhne ji hlasovací skupině pro hlasování. Počká, dokud v daném kroku není rozhodnuto, o které transakci se v daném kroku bude hlasovat. Pokud se hlasuje o jiné transakci, než kterou stránka navrhla v předchozím kroku, tak ji stránka dá svůj hlas. Poté je transakce vyjmuta z fronty nerozhodnutých transakcí a přidána do fronty rozhodnutých transakcí. Pokud transakce mění data, která stránka obsahuje, tak stránka počká, až obdrží všechny hlasy hlasovací skupiny a pokud je alespoň jeden hlas záporný, transakci zruší. Pokud ovšem všechny stránky hlasovaly kladně, tak ji stránka zařadí do řady potvrzených transakcí a provede ji. Na konci se přejde do dalšího kroku.

2. Přijímání nových transakcí z hlasovací skupiny a jejich uložení do fronty nerozhodnutých transakcí.

3. Přijímání hlasů právě rozhodovaných transakcí a uložení hlasu do kolekce přijatých hlasů.

Algoritmus "Many-at-a-time"

U prvního algoritmu, který pro transakce hlasuje sekvenčně, může nastat problém, protože při větším počtu transakcí může dojít k vytvoření velké fronty nerozhodnutých transakcí. Tento algoritmus řeší daný problém tím, že umožňuje navrhovat transakci a hlasovat pro sekvenci transakcí v jednom kroku. U tohoto algoritmu jsou poslední dvě části vykonávány stejně jako u algoritmu předchozího, a proto je popsána pouze ta část, ve které se algoritmus liší od předchozího:

Pokud není fronta nerozhodnutých transakcí prázdná tak stránka hlasuje pro sekvenci tvořenou všemi transakcemi ve frontě nerozhodnutých transakcí a navrhne tuto sekvenci skupině pro hlasování. Počká, dokud v daném kroku není rozhodnuto, o které sekvenci transakcí se bude hlasovat. Po rozhodnutí dá stránka hlas každé transakci v sekvenci, o které se v daném kroku hlasuje (pouze pokud se nejedná o sekvenci, kterou stránka předtím navrhla). Poté algoritmus přesune transakce v sekvenci z fronty nerozhodnutých transakcí do fronty transakcí rozhodnutých. Jestliže v sekvenci existuje transakce upravující data, jež stránka obsahuje, počká stránka nejdříve na všechny hlasy z hlasovací skupiny. Pokud odpověděli všichni ano a transakce upravuje data stránky, přejde do potvrzeného stavu a transakce se provede. Pokud však obdržela transakce alespoň jeden záporný hlas, transakce se zruší. Na konci se přejde do dalšího kroku.

2.1.2 Využití

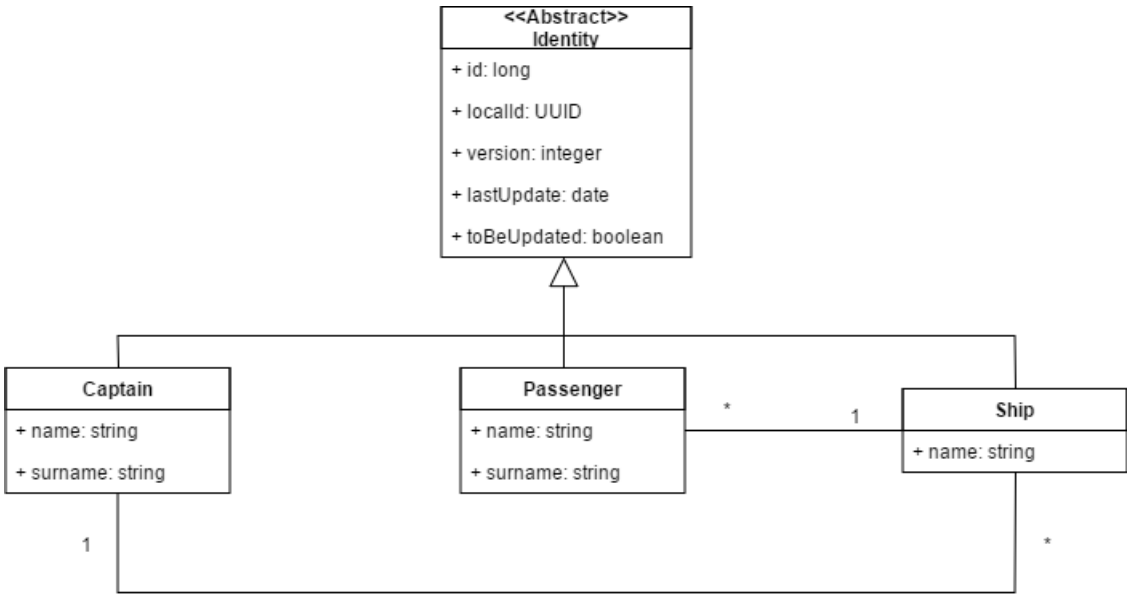
Jak již bylo zmíněno dříve, algoritmy z tohoto článku nelze aplikovat, protože neřeší synchronizaci dat bez možného odpojení od sítě. Dalším problémem je, že žádná ze stránek neobsahuje všechna data a klient by tak nevěděl, odkud si má svá data stáhnout. Pokud by však v budoucnu byla serverová část systému rozdělena na více částí (počítačů), mohl by být použit pro synchronizaci přijatých transakcí algoritmus pro validaci transakcí z tohoto článku.

3 Návrh

Navrhnout a následně vytvořit prototypovou implementaci knihovny, která řeší tento problém, je hlavním cílem této diplomové práce. V této kapitole je popsán i návrh ukázkové implementace systému, ve kterém je tato knihovna použita, aby bylo vysvětleno, jak lze během vývoje knihovnu použít a také jak vyřešit problémy, které knihovna neřeší.

3.1 Vzorový doménový model

Pro ukázku funkčnosti navrhované knihovny je použit jednoduchý doménový model znázorněný na obrázku 2.

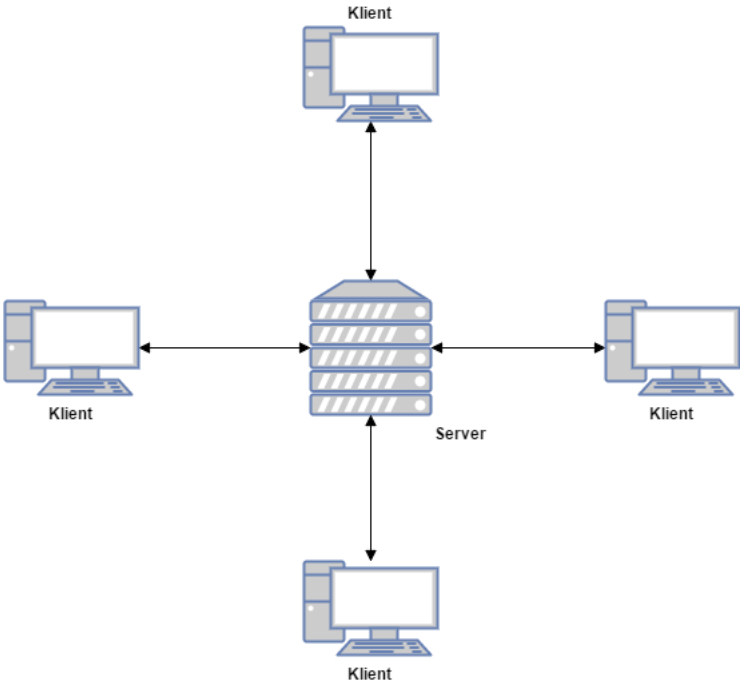


Obrázek 2: Vzorový doménový model

Všechny entitní třídy v projektu, který používá navrhovanou knihovnu, musí implementovat rozhraní `BaseDomain`. Ve vzorové implementaci toto rozhraní implementuje abstraktní třída `Identity`, ze které dědí všechny ostatní entitní třídy a tranzitivně tak implementují dané rozhraní. Důvod použití rozhraní `BaseDomain` a účel atributů, které toto rozhraní vyžaduje, je vysvětlen v kapitole **3.6.3 Aplikace návrhového vzoru Optimistic Offilne Lock**.

3.2 Architektura

Vzorový systém bude rozdělen na serverovou a klientskou část. Server je pasivní část systému, která reaguje na žádosti klientů. Klient je aktivní částí sytému, která posílá žádosti na server a zpracovává jeho odpověď. Tento typ architektury se nazývá *Klient-server* (viz Obrázek 3). Podrobnější návrh každé z obou částí je popsán v dalších kapitolách. Navrhovaná knihovna bude rozdělena na tři části: základní, klientskou a serverovou. Klientská část knihovny je použita v klientské části vzorového systému a serverová část knihovny v serverové části vzorového systému. Základní část knihovny obsahuje třídy, které jsou společné pro klientskou i serverovou část knihovny, aby nedocházelo k duplikaci kódu. Serverová i klientská část knihovny jsou na základní části knihovny závislé.



Obrázek 3: Architektura Klient-server

Komunikace mezi serverem a klienty bude probíhat podle architektury REST. Tato architektura byla zvolena pro svou jednoduchost, nenáročnost a široké rozšíření mezi vývojovými nástroji. Navíc není architektura REST závislá na konkrétním programovacím jazyku či technologii, a proto mohou být v budoucnu implementovány další typy klientských částí, a to v libovolném programovacím jazyce nebo na libovolné platformě bez jakýchkoliv zásahů do serverové části.

3.2.1 Architektura REST

REST je architektura, která umožňuje přistupovat k datům na určitém místě pomocí různých protokolů, nejčastěji je ovšem používán protokol HTTP. Tento protokol je použit i ve vzorové implementaci systému. Architekturu REST v roce 2000 navrhl a popsal Roy Fielding v rámci disertační práce *Architectural Styles and the Design of Network-based Software Architectures*[5].

Metody pro přístup k datům

REST implementuje čtyři metody, které jsou reprezentovány zkratkou CRUD (Create, Read, Update a Delete). V HTTP protokolu jsou tyto metody implementovány pomocí odpovídajících metod tohoto protokolu:

GET (Read) - Základní metoda pro přístup ke zdrojům. Slouží pro načtení seznamu zdrojů nebo obsahu jednoho zdroje.

POST (Create) - Slouží pro vytvoření dat. Ve chvíli volání metody není znám přesný identifikátor (zdroj ještě neexistuje). Proto se pro tuto metodu používá domluvený společný identifikátor.

DELETE (Delete) - Metoda pro smazání zdroje. Volání je obdobné jako u metody GET. Někdy je nahrazena metodou POST s parametrem.

PUT (Update) - Operace pro změnu zdroje. Provádí se volání určitého URI konkrétního zdroje, který má být změněn. Nová hodnota se předává v těle požadavku.

Klient nepracuje přímo se zdroji, ale s jeho reprezentací. Nejčastěji se využívají formáty JSON a XML. Zdroje však mohou být reprezentovány formáty jako PDF, SVG, JPEG, PNG atd. Formát je obvykle definován v URI nebo v HTTP hlavičce *Content-Type*. Vzorová implementace pro přenos dat používá formát JSON.

Systém, který implementuje RESTful architekturu musí splňovat tato omezení:

- **Klient-server** - Omezení popisuje rozdělení systému na serverovou a klientskou část. Oddělením uživatelského rozhraní od správy databáze získáme lepší přenositelnost systému napříč platformami a zlepšíme škálovatelnost zjednodušením serverové části. Největší výhodou tohoto oddělení je, že mohou být vyvíjeny obě části systému nezávisle na sobě.

- **Bezstavovost** - Bezstavovost udává, že každý požadavek od klienta musí obsahovat všechny informace potřebné k pochopení požadavku a nesmí spoléhat na jakékoliv informace uložené na serveru. Díky bezstavovosti dosáhneme lepší viditelnosti, spolehlivosti a škálovatelnosti. Viditelnost bude zlepšena, protože systém nemusí zjišťovat další informace o požadavku. Zlepšení spolehlivosti spočívá v zjednodušeném zotavení systému z částečných poruch. Škálovatelnost bude dále zlepšena z toho důvodu, že server nemusí mezi požadavky uchovávat data, čímž může rychleji uvolňovat zdroje. Bezstavovost zjednodušuje implementaci, jelikož server nemusí spravovat používání zdrojů napříč požadavky. Toto omezení s sebou však nese i svou nevýhodu, která spočívá ve zvýšení zatížení sítě, které je způsobeno opakovaným zasíláním redundantních dat, protože na serveru nemohou být uchována žádná data společná pro skupinu požadavků.

- **Vyrovnávací paměť (cache)** - Hlavním účelem použití vyrovnávací paměti je zlepšení efektivity sítě. Toto omezení požaduje, aby data v odpovědi serveru byla implicitně nebo explicitně označena jako *cachable* (mají se uložit do vyrovnávací paměti) nebo *non-cachable* (nemají se uložit do vyrovnávací paměti). Výhodou tohoto omezení je potencionální eliminace některých interakcí mezi klienty a serverem a tím vylepšení efektivity, škálovatelnosti a výkonu systému. Nevýhodou může být snížení spolehlivosti, pokud klient použije příliš zastaralá data ve vyrovnávací paměti, místo aby získal nejnovější data ze serveru.

- **Jednotné rozhraní** - Architektura REST klade důraz na jednotné rozhraní mezi částmi systému. Díky jednotnému rozhraní se zjednoduší celková architektura systému a zlepší se její viditelnost. Nevýhodou může být snížení efektivity, protože jsou informace mezi komponentami posílány standardizovanou formou.

- **Vrstvený systém** - Vrstvený systém umožňuje architektuře sytému, aby byla rozdělena do hierarchických vrstev tak, že každá z vrstev smí znát pouze tu vrstvu, se kterou bezprostředně komunikuje. Snížíme tím složitost systému a zvýšíme nezávislost jednotlivých vrstev. Nevýhodou vrstveného systému je zvýšení režie a latence.

- **Kód na vyžádání** - REST architektura umožňuje klientovi rozšířit svou funkcionalitu stažením a spuštěním kódu ve formě skriptů. Díky tomuto omezení můžeme zjednodušit klientskou část systému tím, že omezíme počet předem definovaných funkcí, a zároveň vylepšit rozšiřitelnost systému. Nicméně tím také snížíme viditelnost systému, a proto je toto omezení jako jediné volitelné.

Navrhovaný systém splňuje všechna tato omezení kromě omezení kódu na vyžádání, které je ale volitelné, a proto splňuje požadavky kladené na RESTful architekturu.

3.3 Zabezpečení komunikace

Při komunikaci klientské části systému s částí serverovou je vhodné zajistit utajení, integritu a autentizaci posílaných dat, aby nedocházelo k odcizení nebo zfalšování dat. Utajená komunikace je takový způsob přenosu dat, při kterém cizí posluchač naslouchající na přenosovém médiu nerozumí významu přenášených informací a chrání tak komunikaci před odposloucháváním. Zajištěním integrity posílaných dat dáváme příjemci jistotu, že data nebyla na cestě nikým změněna. Autentizací se rozumí jednoznačné určení odesílatele dat přistupujícímu k serveru.

Utajení posílaných dat lze zajistit šifrováním komunikace, a to použitím HTTPS protokolu, který rozšiřuje protokol HTTP. Protokol HTTPS pro šifrování používá kryptografický protokol TLS nebo jeho předchůdce SSL. Organizace IETF[17], která vyvíjí a podporuje internetové standardy, označila šifrovací protokol SSL za zastaralý a nedostatečně bezpečný[18], a proto je vhodnější použít pro šifrování jeho nástupce TLS. Protokol TLS navíc kontroluje integritu dat[35] za použití některé z hashovacích funkcí (jako například SHA, MD5) takže při jeho použití je zajištěna i integrita posílaných dat.

Pro zajištění autentizace může být využit přístup s použitím soukromého tokenu. Při přihlášení uživatele do systému obdrží uživatel soukromý token. Tento token je platný pouze po dobu, kdy je uživatel přihlášený k systému a po jeho odhlášení nebo po uplynutí časového limitu je token označen za neplatný a klient si pro další přístup k serveru musí zažádat o nový. Poté je token zasílán s každým požadavkem na server, který pokaždé ověří, zda uživatel, který se identifikoval zaslaným tokenem, má přístup k vyžádaným datům, a podle toho se rozhodne, jestli uživateli přístup k těmto datům povolí nebo odepře. Pokud během používání aplikace není klientská část systému připojena k síti, bude vyžadováno přihlášení až v době, kdy přístup k síti získá.

Pro aplikaci tohoto přístupu lze použít otevřený protokol OAuth (Open Authentication) [36]. Hlavním účelem tohoto protokolu je poskytnout bezpečnou autentizaci a autorizaci uživatelů. Jednou z velkých výhod OAuth je, že se uživatel může přihlásit do aplikace používající tento protokol pomocí účtu jiné služby (například pomocí účtu Facebook, Google, Microsoft atd.), aniž by uživatel aplikaci musel prozrazovat své heslo. Pro uchování jednoduchosti vzorové implementace bylo zabezpečení komunikace opomenuto, avšak v reálném použití by měla být komunikace mezi klientem a serverem zabezpečena.

3.4 Výběr platformy

Knihovna by měla být dostupná pro nejširší možnou skupinu vývojářů, a proto by se při volbě programovacího jazyka mělo přihlížet k současnému trendu. Z tohoto důvodu byly brány v potaz statistika GitHut 2[14], která sbírá informace o programovacích jazycích používaných na platformě GitHub[15] a TIOBE index[13]. TIOBE index je tvořen podle dat získaných z vyhledávačů jako je Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube atd. Tabulka 1 znázorňuje prvních deset programovacích jazyků na TOIBE indexu k únoru 2018 a tabulka 2 obsahuje deset programovacích jazyků seřazených podle počtu poslaného kódu do repozitářů na platformě GitHub ve čtvrtém čtvrtletí roku 2017.

Pozice	Jazyk	Zastoupení
1	Java	14.988%
2	C	11.857%
3	C++	5.726%
4	Python	5.168%
5	C#	4.453%
6	Visual Basic .NET	4.072%
7	PHP	3.420%
8	JavaScript	3.165%
9	Delphi/Object Pascal	2.589%
10	Ruby	2.534%

Tabulka 1: TIOBE index

Pozice	Jazyk	Zastoupení
1	JavaScript	21.108%
2	Python	14.232%
3	Java	11.264%
4	C++	8.098%
5	C#	7.827%
6	PHP	6.384%
7	C	4.676%
8	Ruby	4.417%
9	Shell	4.292%
10	Go	3.769%

Tabulka 2: GitHut 2 statistika

Z tabulek je vidět, že jeden z nejpoužívanějších programovacích jazyků je Java. Tento jazyk je navíc multiplatformní a disponuje širokou škálou různých knihoven, frameworků a nástrojů, a proto pro implementaci navrhované knihovny byla vybrána právě Java.

Vzhledem k povaze řešené problematiky bude klientská část vzorového systému implementována na některé z platforem pro mobilní telefony. Opět je vhodné při volbě mobilní platformy vzít v potaz současný trend mobilních operačních systémů. V tabulce 3 je znázorněn podíl jednotlivých mobilních platforem na trhu mobilních operačních systémů.

Data pro tuto tabulku byla získána z webu StatCounter[19] a jsou platná k únoru 2018.

Operační systém	Zastoupení
Android	74.39%
iOS	19.64%
Ostatní	3.03%
Windows	0.61%
Firefox OS	0.51%
Series 40	0.5%

Tabulka 3: Statcounter - Podíl na trhu mobilních operačních systémů

Z uvedené tabulky je zřejmé, že trhu mobilních operačních systémů jasně dominuje Android a byl proto vybrán jako platforma pro implementaci klientské části vzorové implementace. Díky použití obecné architektury webového rozhraní REST a obecného formátu pro přenos dat (JSON) může být v budoucnu implementována klientská část systému pro další mobilní platformy.

3.5 Serverová část

Hlavními úlohami serverové části systému je poskytování přístupu k datům pomocí RESTful rozhraní a persistentní ukládání všech klientských dat. Navrhovaná knihovna i vzorová implementace využívají framework Spring [16]. Spring je open source aplikační framework pro tvorbu aplikací na platformě Java. Základní funkce frameworku mohou být použity pro jakoukoliv Java aplikaci, ale existuje spoustu rozšíření pro tvorbu webových aplikací, a proto je tento framework používán spíše pro webové aplikace. Ve vzorové implementaci jsou použita rozšíření Spring frameworku *Web MVC*, které slouží pro usnadnění vývoje webových aplikací a *Spring-ORM*, které usnadňuje práci s databází.

3.5.1 Ukládání dat

Pro komunikaci s databází je použito rozhraní *Java Persistence API* (JPA)[8]. JPA je specifikace rozhraní pro platformu Java, které popisuje správu relačních dat a objektově-relační mapování v aplikacích. Jelikož je JPA pouze rozhraní, je potřeba použít v aplikaci některou z jeho implementací. Pro serverovou část byla vybrána implementace *Hibernate ORM* [9].

Vzorová implementace je nastavena pro komunikaci s open source databází *PostgreSQL*, ale díky obecnosti JPA může být zvolená databáze kdykoliv změněna stažením ovladače pro zvolenou databázi a změněním konfigurace v souboru `config.properties`, který se nachází ve složce properties.

3.5.2 Rozhraní

Rozhraní serverové části je vytvořeno pomocí Spring MVC frameworku. Pro vytvoření rozhraní je potřeba nejdříve vytvořit třídu a označit ji anotací `@Controller`. Pokud nebudou žádné metody ve třídě vracet stránky, ale budou vracet objekty, je lepší třídu označit anotací `@RestController`. Použitím anotace `@RestController` odpadá nutnost anotovat každou metodu třídy anotací `@ResponseBody`, která slouží k označení metod, jež vracejí místo webové stránky objekty. Příklad třídy definující rozhraní ve Spring frameworku je znázorněn ve výpise 1.

```
@RestController
public class ProductController {

   @RequestMapping(method = { RequestMethod.GET }, path = "/products/{id}")
   public Product getProduct(@PathVariable long productId);
      return productDAO.getProduct(productId);
   }
```

Výpis 1: Příklad rozhraní ve Spring frameworku

V tomto příkladu je definována třída s jednou metodou, která má vracet produkt s daným identifikátorem zaslaným v požadavku. Každá metoda, která má obsluhovat zaslaný požadavek, musí být označena anotací `@RequestMapping`, ve které musí být definována HTTP metoda a URL adresa, které má metoda obsluhovat. Pokud metoda přijímá v požadavku parametry, pak musí být tyto parametry označeny. Jestliže se jedná o parametr z hlavičky požadavku, musí být označen anotací `@RequestHeader`, a navíc musí být definován také klíč, pod kterým se hodnota parametru v hlavičce vyskytuje. Jestliže je však parametr brán z těla požadavku, musí být označen anotací `@RequestBody`. V příkladu je parametr brán přímo z URL adresy, ve které je označen pomocí složených závorek a poté je označen v parametru metody anotací `@PathVariable`. Podrobnější popis nastavení webového rozhraní ve frameworku Spring MVC lze nalézt v oficiální dokumentaci frameworku [1]. Spring automaticky převádí tělo odpovědi do formátu JSON (pomocí knihovny *Jackson* [22]), ale jde nastavit i jiný formát. V navrhované knihovně je vytvořeno rozhraní `EntityController`, které by měla implementovat každá třída poskytující webové rozhraní pro operace s entitami. Tato knihovna také obsahuje rozhraní `MainEntityController`, jež by měla implementovat třída, která bude poskytovat webové rozhraní pro práci s hlavními entitami (hlavní entita je popsána dále v kapitole **3.5.5 Extraktor podgrafu**).

3.5.3 Serializace objektů při komunikaci

Při komunikaci mezi klientskou a serverovou částí systému jsou data serializovaná do formátu JSON. Serializace je proces ukládání objektů do formátu, který může být uložen na disk nebo poslán po síti. Pro serializaci a deserializaci dat je ve vzorovém systému a knihovně použita knihovna Jackson[22].

Při serializaci entit může dojít k problému se zacyklením u entit, které mají obousměrnou vazbu. Mějme například dvě entity: e_1 a e_2, kde obě entity mají vazbu na tu druhou. Program začne serializovat entitu e_1, ve které najde odkaz na entitu e_2 a začne ji tedy také serializovat. Přičemž v této entitě najde zpětný odkaz na entitu e_1 a dostane se tím do nekonečné smyčky, která po čase způsobí zhroucení programu. Řešením je použít na zpětné vazby anotaci `@JsonIgnore`. Vlastnosti a vazby označeny touto anotací jsou poté během serializace a deserializace ignorovány a nedojde tedy k zacyklení. Aby mohla být zpětná vazba obnovena při deserializaci objektů, je potřeba přidat místo ignorovaných vazeb identifikátor entity, na kterou se zpětná vazba odkazovala. Po deserializaci lze poté pomocí tohoto identifikátoru získat odkazovanou entitu z databáze a přiřadit ji zpátky dané entitě.

[1] https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-controller

Další problém, který musel být vyřešen, byla serializace entit získaných z databáze pomocí *HibarnateORM*. Pokud Hiberate z databáze stáhne entity, které mají vazbu na další entity, jež z databáze staženy nebyly, nenechá tyto entity prázdné, ale nahradí je svými zástupnými objekty. Při pokusu o získání těchto entit je Hibernate dodatečně z databáze stáhne. K tomu, aby byly entity staženy, musí být program stále v kontextu Hibernatu, jinak nastane chyba. Když se program snaží serializovat data, která chce poslat klientovi, může narazit na zmíněné zástupné objekty a pokusí se je také serializovat. Tyto objekty by ale serialiovány být neměly, protože nejsou součástí dat, které mají být zaslány klientovi. Navíc při serializaci již není program v kontextu Hibernatu, a proto nastane chyba.

Daný problém je vyřešen použitím zásuvného modulu pro knihovnu *Jackson* nazvanou *Jackson datatype hibernate* [23]. Pro přidání tohoto zásuvného modulu musí být nejdříve vytvořena třída, která rozšiřuje třídu `ObjectMapper` a v konstruktoru nově nové třídy musí být zavolána metoda `registerModule`, které je v parametru předán nový objekt typu `Hibernate5Module`. V knihovně je tato třída nazvána `HibernateAwareObjectMapper`. Poté musí být v kontextu Springu tento objekt nastaven jako výchozí třída pro serializaci a deserializaci objektů. Příklad tohoto nastavení je zobrazen ve výpise 2.

```
<mvc:annotation-driven>
   <mvc:message-converters>
      <bean
         class="org.springframework.http.converter.json.
            MappingJackson2HttpMessageConverter">
         <property name="objectMapper">
            <bean class="com.vsb.dsslib.server.
               dao.HibernateAwareObjectMapper" />
         </property>
      </bean>
   </mvc:message-converters>
</mvc:annotation-driven>
```

Výpis 2: Nastavení výchozí třídy pro serializaci a deserializaci ve frameworku Spring

Dále je nutné zajistit, aby nebyly serializovány atributy entit obsahující prázdné hodnoty objektů nebo výchozí hodnoty primitivních proměnných. Tyto atributy by neměly být serializovány, protože nemají pro klienta žádnou vypovídající hodnotu. Knihovna Jackson umožňuje nastavit pravidla pro zařazování atributů do serializace podle jejich hodnot, a to pomocí metody `setSerializationInclusion`. Pro vyřazení atributů s prázdnými hodnotami musí být tato metoda zavolána s parametrem `Include.NON_NULL` a pro vyřazení atributů s výchozími hodnotami s parametrem `Include.NON_DEFAULT`. Pokud je u některých primitivních proměnných potřeba, aby byly serializovány i jejich výchozí hodnoty, lze je označit anotací `@JsonInclude`, která zajistí, že budou pokaždé serializovány.

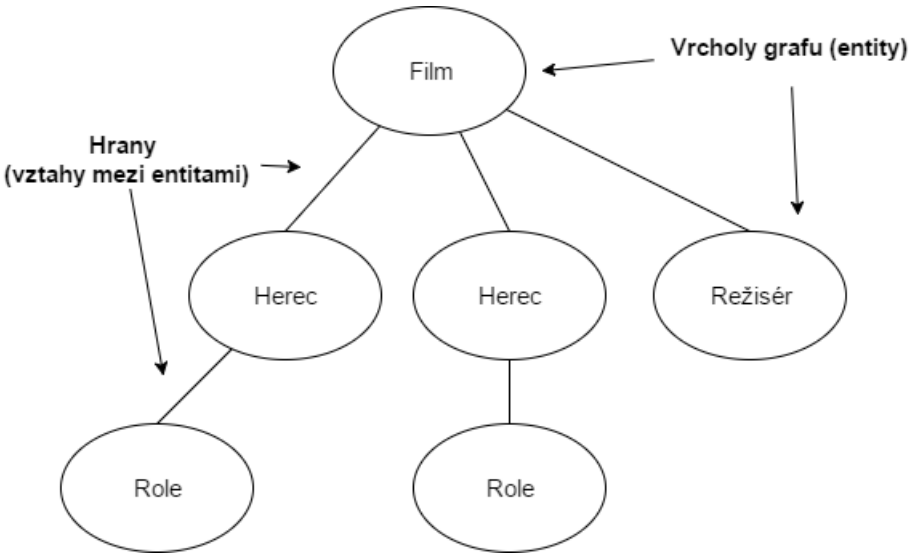
Při serializaci musí být také nastaven správný formát data a času. Ten musí musí být stejný jak pro serverovou, tak klientskou část systému, jinak by při serializaci a deserializaci dat nastávaly chyby. Datový formát pro knihovnu Jackson je nastaven pomocí metody `setDateFromat`, která v argumentu přijímá objekt abstraktní třídy `DateFormat`. V knihovně je této metodě dodán objekt typu `SimpleDateFormat`, který rozšiřuje třídu `DateFormat`. Datový formát je načítán ze souboru `config.properties` (tento soubor musí být umístěn ve složce `properties` v kořenové složce zdrojů projektu), kde musí být nastaven pod klíčem `dataFormat`. Ve vzorové implementaci je použit datový formát ve tvaru: "den.měsíc.rok hodina:minuty:sekundy". Pro účel vzorové implementace je daný časový formát dostačující, ale v reálném nasazení by měl být nastaven jemnější formát, protože na server může chodit v rámci jedné sekundy více požadavků.

U některých atributů entitních tříd může být nastaveno, aby nebyly ukládány do databáze (v navrhované knihovně se jedná například o atribut `toBeUpdated`, u kterého nemá smysl jej do databáze ukládat, protože je jeho hodnota platná jenom v rámci dané transakce). Tyto atributy jsou v JPA označeny pomocí anotace `@Transient`. Při použití této anotace však nastává problém při serializaci anotovaného atributu, protože zásuvný modul *Jackson datatype hibernate* knihovny Jackson ve výchozím nastavení atributy označené touto anotací během serializace a deserializace ignoruje. Aby tyto atributy ignorovány nebyly, je potřeba objektu zodpovědnému za serizalizaci vypnout pomocí metody `disable` vlastnost serializace nazvanou `USE_TRANSIENT_ANNOTATION`. Po provedení tohoto nastavení nebudou atributy označené anotací `@Transient` ignorovány.

Všechna nastavení, která řeší popisované problémy, jsou již nastaveny ve třídě `HibernateAwareObjectMapper` (s výjimkou zacyklení entit), která je součástí knihovny a uživatel knihovny proto nemusí tyto problémy už řešit. Tuto třídu stačí pouze zaregistrovat do kontextu frameworku Spring jako výchozí třídu pro serializaci a deserializaci objektů tak, jak bylo popsáno výše.

3.5.4 Snížení zatížení sítě

Při posílání dat mezi serverovou a klientskou částí systému je vhodné optimalizovat tuto komunikaci tak, aby co nejméně zatěžovala síť. Prvním krokem, jak komunikaci optimalizovat, je posílání pouze těch dat, která klientská část opravdu potřebuje, a jsou pro ni relevantní. Na entity a vztahy mezi nimi může být pohlíženo jako na graf, v němž jsou jednotlivé entity vrcholy a jejich vzájemné vztahy jsou hrany (příklad takového grafu je zobrazen v obrázku 4). Na serveru je tedy uložen kompletní graf všech entit a je potřeba poslat klientovi podgraf entit, které jsou pro něj relevantní. Pro tento účel byla v knihovně navržena komponenta nazvaná extraktor, která je popsána v kapitole **3.5.5 Extraktor podgrafu**.



Obrázek 4: Graf entit

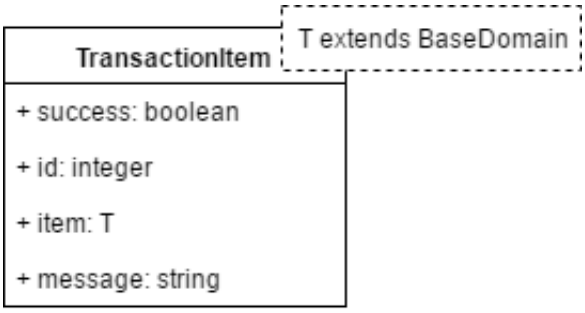
Dále je komunikace optimalizována tak, že serverová část neposílá klientovi entity, které od posledního poslání dat klientovi nebyly změněny. Každá entita v sobě má uloženou informaci o tom, kdy byla naposledy upravena (atribut `lastUpdate`). Klient v každém požadavku o zaslání dat na server zašle také informaci o tom, kdy naposledy úspěšně o tato data žádal. Server poté projde všechny entity, které mají být klientovi poslány, a u těch, u kterých od posledního zaslání neproběhly žádné změny, smaže všechna data a ponechá pouze identifikátor entity. Tyto entity jsou dále označeny příznakem (atribut `toBeUpdated` je nastaven na zápornou hodnotu), který dává klientovi informaci o tom, že entita nebyla změněna a že ji nemá ukládat do své databáze. Pro ještě lepší optimalizaci sítě by se zmíněné entity nemusely posílat vůbec, ale to by přineslo určité problémy:

Vztahy mezi entitami - Entity mezi sebou mají různé vztahy a pokud by graf entit zasílaný klientovi byl nesouvislý, jednotlivé entity by mohly ztratit informaci o vztazích k jiným entitám.

Mazání zastaralých entit - Pokud bychom klientovi zasílali pouze změněné entity, nevěděl by, zda nemá ve své databázi uložené zastaralé entity, které byly na serveru už smazány. Mazání zastaralých entit na klientské části systému je popsáno v kapitole **3.6.10 Mazání zastaralých entit**.

Pro snížení počtu zasílaných požadavků klienta na server jsou entity posílány v transakcích místo toho, aby byly posílány jednotlivě. Každá transakce obsahuje tři kolekce: kolekci nově vytvořených entit, kolekci upravených entit a kolekci entit smazaných. Každé položce v těchto kolekcích je před posláním na server přidělen v rámci transakce jedinečný identifikátor.

V navrhované knihovně jsou transakce reprezentovány třídou `Transaction`. Server transakci zpracuje a pošle klientovi odpověď (třída `TransactionResponse`) s kolekcí výsledků operací nad zaslanými entitami. Každý výsledek je reprezentován třídou `TransactionItem`, jejíž třídní diagram je znázorněna na obrázku 5. Tato třída je generická a lze ji použít pro jakoukoliv třídu implementující rozhraní `BaseDomain`. Díky použití generického typu není potřeba definovat třídu pro každý entitní typ zvlášť.



Obrázek 5: Třída TransactionItem

Z diagramu lze vidět, že každá položka odpovědi transakce obsahuje:

Příznak úspěchu - Tento příznak je nastaven na kladnou hodnotu (true) tehdy, jestliže operace nad danou entitou proběhla v pořádku a změny se na serveru uložily do databáze. V opačném případě je nastaven na zápornou hodnotu (false), aby klient dostal informaci o chybě a mohl na ni reagovat.

Identifikátor položky - Identifikátor, který byl objektu přidělen před posláním na server, je posílán zpět klientovi, aby si mohl spárovat položky, které poslal s vrácenými položkami.

Entitu - Objekt vytvořený při operaci. Tato položka je vyplněna při vytváření nových objektů, aby klient mohl uložit globální identifikátor přidělený při vytvoření objektu do své databáze. Je také vyplněna, pokud klient poslal požadavek pro úpravu dat, která však byla zastaralá, a je v ní poslána entita s nejnovější verzí, aby se klient mohl rozhodnout, jak daný konflikt vyřeší.

Zprávu - Zpráva pro klienta o výsledku operace. Zpráva je důležitá jenom v případě, že nastala při operaci nějaká chyba.

Zavedením transakcí je navíc značně zjednodušeno rozhraní serverové části systému, protože nemusí poskytovat pro každou entitu jednotlivé operace pro vytvoření, upravení a smazání dat, ale pouze operaci pro přijetí transakce.

3.5.5 Extraktor podgrafu

Jak již bylo popsáno dříve, funkce této komponenty spočívá ve výběru pro klienta relevantního podgrafu entit. Na začátku grafu se nachází hlavní entita, která je pro uživatele relevantní. K této hlavní entitě jsou poté přidávány do grafu další entity, které je potřeba načíst a které mají k hlavní entitě vztah.

Prvním přístupem, jak extraktor implementovat, bylo využití frameworku *Hibarnate ORM*, konkrétně třídy `org.hibernate.persister.entity.AbstractEntityPersister`. Tato třída umí poskytnout informace o dané entitě jako je seznam atributů třídy, třídu, ze které entita dědí, klíč entity atd. Tento přístup měl však několik nežádoucích vlastností:

1. **Ztráta obecnosti JPA** - Třída `AbstractEntityPersister` je specifická pro Hibernate ORM implementaci JPA. Pokud by byla použita, knihovna by se tak stala závislá na Hibernate ORM frameworku a nemohla by ho jednoduše zaměnit za jinou implementaci.

2. **Občasná nefunkčnost** - Některé metody třídy `AbstractEntityPersister` nefungují vždy správně. Například metoda `getMappedSuperclass` nevrací nadtřídu dané entity, ale prázdnou hodnotu, přestože je v dokumentaci jasně popsáno, že má vracet nadtřídu entity.

Kvůli zmíněným důvodům bylo upuštěno od použití Hibernate frameworku pro tuto komponentu. Dalším přístupem pro implementaci extraktoru bylo využití reflexe. Reflexe je schopnost programovacího jazyka zjistit informace o třídách, jejich proměnných a metodách za běhu programu, a to bez nutnosti znát tyto informace v době kompilace programu. Jedná se o poměrně pokročilou techniku, kterou by měli používat zkušenější programátoři. Součástí jazyka Java je rozhrání Java Reflection API[11], které slouží pro práci s reflexí. Během implementace tohoto přístupu však začalo být jasné, že výsledné řešení bude na daný úkol náročné jak paměťově, tak časově a celkově zbytečně složité.

Po důkladnějším průzkumu problému nakonec bylo zvoleno řešení pomocí entitních grafů v JPA. Entitní grafy byly v JPA představeny ve verzi 2.1 a jedná se o funkci, která programátorovi umožňuje dopředu vytvořit plán získávání dat z databáze a poté jej kdykoliv použít pro získání konkrétního grafu entit. V JPA existují dva typy těchto entitních grafů:

Statické - Statické grafy (v JPA se označují jako pojmenované) se definují přímo v entitní třídě v době kompilace pomocí anotace `NamedEntityGraph`. V atributech se pak musí definovat jméno tohoto grafu a entity, které mají být do grafu zahrnuty. Můžeme zde také definovat podgrafy, které chceme do výsledného grafu zahrnout. Příklad statického entitního grafu a jeho použití při získávání dat z databáze je znázorněn ve výpise 3.

```
@Entity
@NamedEntityGraph(name = "graph.movie.actors.roles",
    @NamedAttributeNode(value = "actors", subgraph = "actorRoles"),
    subgraphs = @NamedSubgraph(name = "actorRoles", attributeNodes =
        @NamedAttributeNode("roles")))
public class Movie {
   @ManyToMany
    private Set<Actor> actors;
    ...
}
EntityGraph graph = entityManager.getEntityGraph("graph.movie.actors.roles");
Map hints = new HashMap();
hints.put("javax.persistence.fetchgraph", graph);
entityManager.find(Movie.class, movieId, hints);
```

Výpis 3: Statický entitní graf a jeho použití

Dynamické - Dynamické grafy se mohou oproti statickým definovat za běhu programu. Nejdříve je potřeba vytvořit objekt typu `EntityGraph` a specifikovat, pro jaký entitní typ bude graf vytvořen. Poté mohou být přidávány podgrafy (třída `Subgraph`) a postupně tak vytvořit výsledný graf. Ve výpise 4 je znázorněno vytvoření dynamického entitního grafu a jeho použití při získávání dat z databáze.

```
EntityGraph<Movie> graph = entityManager.createEntityGraph(Movie.class);
Subgraph<Actor> itemGraph = graph.addSubgraph("actors");
itemGraph.addAttributeNodes("roles");

Map<String, Object> hints = new HashMap<String, Object>();
hints.put("javax.persistence.fetchgraph", graph);
entityManager.find(Movie.class, movieId, hints);
```

Výpis 4: Dynamický entitní graf a jeho použití

V knihovně jsou využity pouze dynamické grafy, a to z důvodu zachování obecnosti knihovny. Použitím entitních grafů se zachová obecnost rozhraní JPA (implementace nebude závislá na konkrétní implementaci JPA) a celkově se usnadní vývoj extraktoru.

Konfigurace extraktoru podgrafu

Před použitím extraktoru je potřeba zaregistrovat třídy `EntityExtractor` a `ConfigParser` jako komponenty v kontextu Spring frameworku. Nejjednodušší způsob, jak toho dosáhnout, je nastavit v aplikaci automatické skenování balíčku `com.vsb.dsslib.server.dao`. Vzhledem k rozšířenosti frameworku Srping existuje na internetu spoustu návodů, jak automatické skenování nastavit[2], a proto zde nebude toto nastavení popsáno.

Extraktor se konfiguruje pomocí XML souboru, jehož název a cesta k němu jsou konfigurovány v souboru `config.properties` pod klíčem *configFilePath*. Tento soubor se musí nacházet ve složce properties, která by se měla nacházet v kořenové složce zdrojů projektu. Formát XML byl zvolen, protože je v něm možné jednoduše zachytit hierarchie jednotlivých elementů.

Podle specifikace jazyka XML[10] by měl každý validní XML soubor začínat značkou `<?xml>`, která obsahuje XML deklaraci. V této deklaraci je specifikována použitá verze jazyka a použité kódování. Po XML deklaraci musí být v konfiguraci specifikovaný hlavní element `<root>`, který musí obsahovat atribut *class*, v jehož hodnotě se specifikuje celá cesta ke třídě hlavní entity, včetně jména třídy. Uvnitř tohoto hlavního elementu se může nacházet libovolný počet značek `<field>`, které musí obsahovat atribut *name*. Jeho hodnota pak musí odpovídat názvu proměnné objektu, který značka reprezentuje. Každá značka `<field>` může uvnitř sebe dále obsahovat další značky stejného typu. Tato konfigurace musí odpovídat grafu entit, který chceme synchronizovat s klientskou částí systému. Ve výpise 5 je znázorněn příklad konfigurace extraktoru.

```
<?xml version="1.0" encoding="UTF-8"?>
<root class="com.corporate.domain.Customer">
   <field name ="orders">
      <field name ="items"></field>
   </field>
   <field name ="favourites">
   </field>
</root>
```

Výpis 5: Ukázka konfigurace extraktoru

Pro použití extraktoru v projektu stačí vytvořit instanci třídy `EntityExtractor`. Pro získání relevantních dat klienta pak stačí nad instancí zavolat metodu `getData` a v parametru metody předat identifikátor hlavní entity, pro kterou je potřeba získat data.

[2] https://www.boraji.com/spring-4-auto-component-scanning-example

Čtení konfigurace

Pro čtení konfigurace byla v knihovně vytvořena třída nazvaná `ConfigParser`, která má za úkol přečíst strukturu a obsah konfigurace a vytvořit podle ní entitní graf. Programovací jazyk Java nabízí dva způsoby, jak přistupovat ke struktuře a obsahu XML souboru[12]:

1. **Document Object Model (DOM)** - Celý XML dokument je nahrán do paměti a vytvoří se stromová reprezentace dokumentu, se kterou pak může programátor pracovat. Při použití tohoto přístupu můžeme dokument číst, upravovat i ukládat.

2. **Simple API for XML (SAX API)** - SAX je založený na událostech při procházení dokumentu. Při tomto procházení je pro každý důležitý prvek dokumentu (jako je počáteční a koncová značka, znaková data, komentář apod.) vyvolána událost, kterou programátor může obsloužit. V parametrech události jsou přitom předány informace, jako třeba název elementu, text obsažený ve znakových datech apod. Tento přístup je vhodnější použít u větších dokumentů, protože nenačítá celý dokument do paměti, ale pouze ho sekvenčně čte a je díky tomu i rychlejší. Při použití SAX můžeme XML dokument pouze číst, nikoliv však upravovat a ukládat.

Pro účely čtení konfigurace byl použit první z přístupů (DOM), a to z důvodu snadnějšího zacházení s hierarchií jednotlivých elementů. Nepředpokládá se, že konfigurační soubory budou natolik velké, aby paměťová a časová náročnost SAXu byla znatelná.

Čtení konfigurace a vytvoření entitního grafu ve třídě `ConfigParser` probíhá následovně:

1. Konfigurační soubor je načten do paměti.

2. Z hlavního elementu dokumentu `<root>` je načtena třída hlavní entity, ze které je vytvořen hlavní objekt grafu (objekt typu `EntityGraph`).

3. Pro každý element `<field>` uvnitř hlavního elementu je podle jména uvnitř atributu *name* vytvořen objekt podgrafu (objekt typu `Subgraph`), který je přiřazen do hlavního objektu grafu.

4. Pokud element `<field>` v předchozím kroku obsahoval další elementy stejného typu, jsou i pro ně vytvořeny objekty podgrafu, které jsou přiřazeny do podgrafu vytvořeném v předchozím kroku. Všechny další elementy uvnitř těchto elementů jsou zpracovány tímto bodem rekurzivně dokud není přečten celý konfigurační soubor.

3.5.6 Konfigurační management serverové části

Sestavení projektu a konfigurace prostředí

Pro sestavování a správu navrhované knihovny a serverové části vzorového systému je použit nástroj Apache Maven[21]. Projekty v Mavenu jsou konfigurovány v souboru `pom.xml` (POM = project object model), který se musí nacházet v kořenovém adresáři projektu. V souboru `pom.xml` je popsán proces sestavování projektu a akce s ním spojené (jako například spouštění testů). Jsou zde také popsány závislosti na externích knihovnách. Při sestavování projektu pak Maven automaticky všechny tyto knihovny stáhne. Použitím tohoto nástroje pro sestavování projektu odpadne závislost na konkrétním vývojovém prostředí, protože všechny informace pro kompilaci a sestavení projektu jsou obsaženy v souboru `pom.xml`.

Před sestavením serverové části vzorového projektu je potřeba nastavit přístupové údaje do repozitáře, kde je knihovna uložena. Kromě jména a hesla je potřeba zadat identifikátor, kterým je v projektu repozitář identifikován (v případě navrhované knihovny musí být id nastaveno na hodnotu *central*). Tyto údaje jsou zadávány do souboru `settings.xml`, který se nachází ve složce `.m2`. Složka `.m2` je vytvořena při instalaci nástroje Maven v domovské složce uživatele systému. Pokud v této složce soubor `settings.xml` není, je třeba jej vytvořit. Příklad konfigurace uvnitř souboru `settings.xml` je zobrazen ve výpise 6.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                          https://maven.apache.org/xsd/settings-1.0.0.xsd">
   <servers>
      <server>
         <id>central</id>
         <username>jmeno</username>
         <password>heslo</password>
      </server>
   </servers>
</settings>
```

Výpis 6: Ukázka konfiguračního souboru pro Maven

Pro nastavení přístupu do Maven repozitáře, kde je knihovna uložena, lze použít nastavení v tomto výpisu a změnit pouze jméno a heslo uvnitř daných značek. Zašifrování hesla a další možnosti konfigurace Mavenu v tomto souboru je popsáno v oficiální dokumentaci Mavenu[3]. Po vyplnění přístupových údajů by měl mít Maven přístup do repozitáře, který obsahuje knihovnu, a serverová část systému si ji při sestavení může z tohoto repozitáře stáhnout.

[3] https://maven.apache.org/guides/mini/guide-configuring-maven.html

Pokud uživatel nemá přístup do vzdáleného repozitáře, kde je knihovna uložena, musí být knihovna manuálně sestavena a uložena do lokálního repozitáře. Projekty v Mavenu jsou sestavovány pomocí příkazů v příkazové řádce. Pro sestavení a nainstalování základní (DSSlib-core) a serverové (DSSlib-server) části knihovny do lokálního repozitáře stačí otevřít příkazovou řádku ve složce s danou části knihovny a zadat příkaz: `mvn install`. Základní část knihovny musí být nainstalována jako první, protože serverová část knihovny je na ni závislá.

Pro sestavení serverové části vzorového projektu je potřeba otevřít příkazovou řádku ve složce s projektem a zadat příkaz: `mvn package`. Maven stáhne všechny potřebné knihovny, zkompiluje veškeré třídy projektu a zabalí je do souboru `DSSserver.war`, který po dokončení sestavování bude ve složce target.

Nastavení databáze, datového formátu a cesty k souboru s konfigurací extraktoru dat se konfiguruje pomocí souboru `config.properties`, ve složce properties, která se nachází v kořenové složce zdrojů projektu (`DSSserver/src/main/resources`). Pokud je potřeba změnit nastavení databáze, stačí změnit hodnoty v tomto souboru. Pokud však chceme změnit typ databáze, je navíc potřeba přidat do projektu ještě knihovnu s ovladačem pro danou databázi.

Nasazení

Serverová část systému může být nasazena na jakýkoliv webový server, který podporuje technologii Java. Během vývoje byl používán webový server Apache Tomcat. Pro nasazení projektu před startem serveru musí být soubor `DSSserver.war`, vygenerovaný při sestavení projektu, zkopírován do složky webapps, která se nachází v kořenové složce serveru. Během startu server automaticky projekt rozbalí a nasadí. Pokud je potřeba nasadit projekt za běhu serveru, může k tomu být využito grafické rozhraní pro správu serveru. Do tohoto rozhraní je možné se dostat zadáním URL adresy: `<adresa_serveru:port>/manager`. V sekci *Deploy* se nachází podsekce *WAR file to deploy*, ve které se vybere WAR soubor s projektem a po stisknutí tlačítka *Deploy* server začne nasazovat projekt na server.

3.6 Klientská část

Klient je aktivní část systému, která má za úkol poskytovat uživatelské rozhraní systému, komunikovat se serverem a uchovávat data ve vyrovnávací paměti. V případě, že klient není připojený k síti, dostupná data načítá z této paměti.

3.6.1 Ukládání dat

Pro ukládání dat používá klientská část systému open source relační databázi SQLite[20]. Tato databáze je již zabudovaná v systému Android a není ji proto potřeba jakkoliv importovat. Hlavními výhodami SQLite jsou malá velikost (celá knihovna databáze má okolo 500KiB a pokud se vynechají volitelné funkce, velikost se sníží až na 300KiB), malé nároky na operační paměť (vystačí si i s cca 110KiB) a podpora pokročilých databázových funkcí, jako jsou například transakce, fulltextové vyhledávání, částečné indexy atd. Díky těmto výhodám je to ideální volba pro zařízení s omezenou velikostí paměti, jako je tomu například u mobilních telefonů. Celá databáze je uchovávána na disku v jediném souboru, jehož název se specifikuje při inicializaci databáze. V ukázkové implementaci je název tohoto souboru definován v konfiguračním souboru *settings.xml* (tento soubor se nachází ve složce `DSSclient/app/src/main/res/values`) pod klíčem *database_file_name*.

Pro objektově-relační mapování je použita knihovna ORMLite. Jedná se o lehkou a jednoduchou knihovnu, která poskytuje standardní funkce ORM nástrojů bez přidané komplexnosti a režie jiných ORM knihoven či frameworků. Mezi hlavní funkce ORMLite, které byly užitečné při vývoji vzorové implementace, patří:

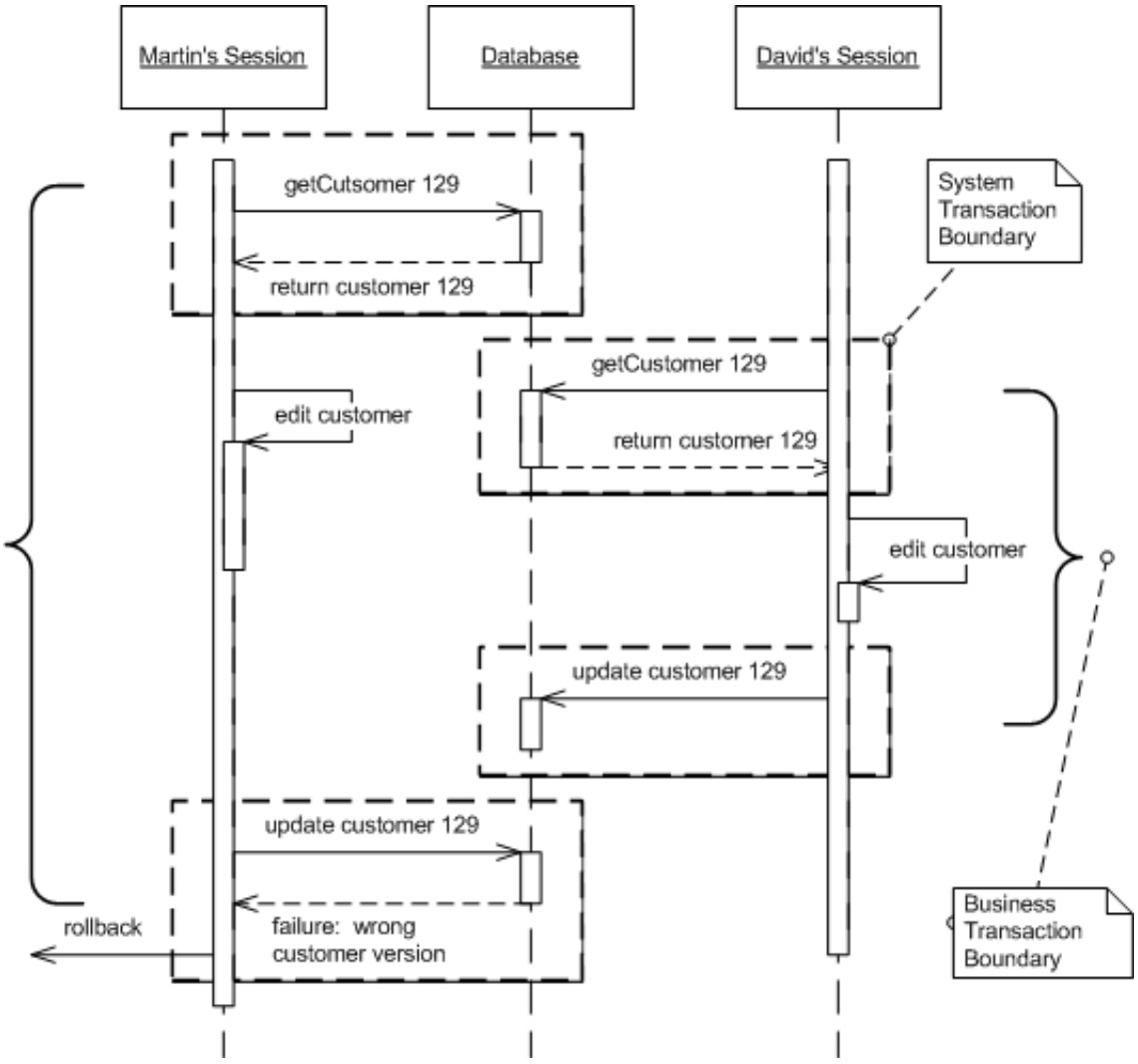
- Rozšiřitelné DAO třídy
- Třídy pro stavbu komplexních dotazů
- Generované třídy pro vytváření a mazání tabulek
- Podpora vtahů mezi entitami

3.6.2 Inspirace návrhu klientské části

Návrh klientské části knihovny byl inspirován návrhovým vzorem *Optimistic Offilne Lock* popsaným v knize *Patterns of Enterprise Application Architecture* [7]. *Optimistic Offilne Lock* slouží k předcházení konfliktu mezi souběžnými business transakcemi tím, že detekuje konflikt a vrátí změny provedené jednou z transakcí zpět. Nejjednodušším způsobem, jak tento konflikt detekovat, je přiřazení verze každému objektu. Při provádění změn se pak verze měněného objektu porovná s verzí tohoto objektu v databázi a změny jsou do databáze promítnuty pouze tehdy, pokud je jeho verze větší.Není vhodné, používat místo čísla verze časové razítko poslední změny, protože systémové hodiny mohou být nespolehlivé, a to hlavně v případě použití více serverů.

Navíc je u každého objektu uchovávána i informace, kdo a kdy objekt naposledy upravil. Tyto informace mohou pro uživatelům být užitečné při řešení konfliktů.

Tento návrhový vzor však nebere v úvahu systém, ve kterém klientská část tohoto systému může fungovat bez připojení k síti, a proto je ve vzorové implementaci aplikován pouze částečně. Slovo *Offline* v názvu totiž neznamená „bez připojení k síti“, ale to, že business transakce jsou řešeny na straně klienta. V knize *Patterns of Enterprise Application Architecture*[7] je business transakce chápána jako série akcí uživatele v systému, které vedou k nějakému výsledku. Například v informačním systému banky může být business transakce složena z přihlášení, výběru účtu, nastavení platby a kliknutí na potvrzovací tlačítko, čímž se platba provede. Příklad fungování návrhového vzoru *Optimistic Offilne Lock* je znázorněno v obrázku 6, který je převzat z webu Martina Fowlera[24]. V tomto obrázku je rozsah business transakcí znázorněn čarami na bocích a rozsah systémových transakcí je znázorněn přerušovanou čarou.

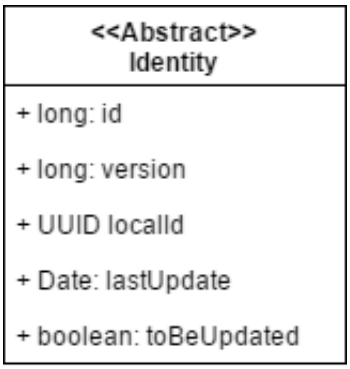


Obrázek 6: Návrhový vzor Optimistic Offilne Lock[24]

3.6.3 Aplikace návrhového vzoru Optimistic Offilne Lock

V knize je popsán krátký příklad použití tohoto návrhového vzoru. Prvním krokem implementace je použití návrhového vzoru *Layer supertype* tak, aby všechny doménové objekty obsahovaly potřebné informace pro řešení konfliktů mezi transakcemi. *Layer supertype* je velice jednoduchý návrhový vzor, který popisuje použití třídy, ze které dědí všechny doménové objekty. Tato nadtřída obsahuje atributy, které mají být společné pro všechny entity. V případě *Optimistic Offilne Lock* by měla tato nadtřída obsahovat identifikátor, číslo verze objektu, informaci o tom, kdo objekt naposledy upravil, a datum a čas poslední modifikace.

Každá entitní třída v systému využívajícím navrhovanou knihovnu proto musí implementovat rozhraní `BaseDomain`, které zajišťuje, že všechny tyto entitní třídy budou požadované informace obsahovat. Abstraktní třída `Identity`, ze které ve vzorové imlementaci dědí všechny entitní třídy (třídní diagram této třídy je znázorněn na obrázku 7), implementuje rozhraní `BaseDomain`.



Obrázek 7: Třída Identity

Z obrázku lze vidět, že v této třídě byla vynechána informace o tom, kdo naposledy objekt upravil. Tato informace není v rozhraní `BaseDao` požadována pro zjednodušení knihovny. Avšak v některých systémech nemusí být tato informace ani relevantní. Naopak jsou zde navíc atributy *localId* a *toBeUpdated*, které jsou rozhraním vyžadovány. Použití atributu *toBeUpdated* bylo popsáno v kapitole **3.5.4 Snížení zatížení sítě**. Důvod použití atributu *localId* je popsán v kapitole **3.6.4 Lokální identifikátor**.

Dále je v příkladu použit návrhový vzor *Data Mapper*. Data Mapper je popsán Martinem Fowlerem[25] jako vrstva mapujících objektů, která přenáší data mezi objekty a databází, přičemž je ponechává vzájemně nezávislé a také nezávislé na mapujícím objektu. Tento návrhový vzor je implementován použitím knihovny pro objektově relační mapování ORMLite. Použití této knihovny je navíc zaobaleno vrstvou DAO objektů. Data access object (DAO) je návrhový vzor, který zapouzdřuje veškerý přístup k datovému zdroji. Výhodou použití tohoto návrhového vzoru je, že zbytek aplikace se stane nezávislý na použité databázi a knihovně pro objektově-relační mapování. V příkladu tento Data Mapper komunikuje s databází pomocí jazyk SQL. Operace mazání a úpravu dat provádí pouze na objektu, který má stejnou verzi (na konci SQL dotazu

je vždy klauzule WHERE version=verze_objektu). Poté je zkontrolován vrácený počet objektů, které byly dotazem změněny a pokud je roven nule, program vyhodí výjimku.

Ve vzorové implementaci je kontrola verze objektů prováděna na serverové straně systému, a pokud tato kontrola selže, je daná položka transakce označena za chybnou a poté co je poslána zpátky, může se klientská část systému rozhodnout, co s danou chybnou položkou bude dělat.

V příkladu je i popsáno, jak lze pomocí tohoto návrhového vzoru předcházet čtení zastaralých dat. Vzhledem k tomu, že klientská část nemusí být vždy připojena k síti, nelze tento mechanismus implementovat a není proto v ukázkové implementaci ani použit.

Na konci příkladu je ještě zmíněn návrhový vzor *Unit of Work*, který je v knihovně použit. Použití tohoto návrhového vzoru v knihovně je popsáno v kapitole **3.6.5 Uchovávání změn entit**.

3.6.4 Lokální identifikátor

Entity na klientské části systému jsou identifikovány pomocí lokálního identifikátoru (`localId`). Aby mohly být entity uloženy do databáze, musí být jednoznačně identifikovány pomocí svého identifikátoru. Na klientské části však nelze použít globální identifikátor entit používaný serverem, protože může nastat situace, kdy je vytvořena nová entita, a to ve chvíli, kdy zrovna není klient připojen k síti. Pokud by tato nová entita měla být uložena do lokální databáze, nastala by chyba, protože by nebyla jednoznačně identifikována. Lokální identifikátor je v knihovně a vzorové implementaci reprezentován pomocí UUID.

UUID (universally unique identifier) někdy také označováno jako GUID (globally unique identifier), je 128 bitové číslo, které je často používáno v počítačových systémech k identifikaci. Pokud je UUID generováno v souladu se standardními metodami, je prakticky unikátní, a to bez závislosti na jakékoliv centrální autoritě nebo koordinace mezi komponentami, které je vytvářejí. Pravděpodobnost vygenerování dvou stejných UUID není nulová, ale je dostatečně blízko nule na to, aby byla zanedbatelná [41]. Existuje několik variant a verzí UUID. Programovací jazyk Java implementuje variantu jedna (také nazývaná Leach-Salz) a nabízí použití všech verzí této varianty. Ve vzorové implementaci je používána verze čtyři, která jako zdroj pro generování UUID používá náhodná čísla.

3.6.5 Uchovávání změn entit

Pro uchovávání informací o vytvořených, změněných a smazaných entitách je v knihovně použit návrhový vzor *Unit of Work*. *Unit of Work*(jednotka práce) je návrhový vzor, který slouží k udržování kolekce objektů, jenž byly modifikovány, smazány nebo vytvořeny v rámci doménových operací, a k jejich následnému uložení do databáze.

Tento návrhový vzor funguje tak, že při vytvoření, upravení nebo smazání entitního objektu je o této akci obeznámena třída `UnitOfWork`. Pro aplikaci těchto změn do databáze je zavolána metoda `commit`, která se postará o provedení změn. Výhodou použití tohoto návrhového vzoru je, že se zmírní zatížení serveru, protože změny nejsou posílány postupně, ale najednou a mohou tak být poslány v jedné transakci, čímž se ušetří režie při posílání dat do databáze.

Hlavním důvodem, proč je tento návrhový vzor použit v navrhované knihovně, je, aby uchovával všechny změny provedené na klientské části systému po dobu, kdy klient nemá přístup k síti a nemůže tedy tyto změny poslat na server. V knihovně je pro tento účel vytvořena abstraktní třída `UnitOfWork`. Pro každý entitní typ, u kterého je potřeba uchovávat změny, musí být vytvořena třída, která rozšiřuje třídu `UnitOfWork`. Každá z těchto tříd musí implementovat metody pro potvrzení změn (`commit`) a uložení, změnu a smazání entit daného typu z lokální databáze. Všechnu ostatní funkcionalitu již třída `UnitOfWork` obsahuje.

V knihovně jsou změny ve třídě `UnitOfWork` uchovávány v kolekci typu `Map`. Tato kolekce uchovává objekty na základě daného klíče. Ve vzorové implementaci je jako klíč použit lokální identifikátor entit. Pokud je do této kolekce uloženo pod stejným klíčem více objektů, bude v ní uložen vždy objekt, který se pod daným klíčem uložil jako poslední. Díky této vlastnosti můžeme na klientovi upravit jednu entitu vícekrát, ale na konci konci bude v kolekci uložený pouze objekt naposledy uložený, jenž bude poté poslán do databáze.

Při aplikaci návrhového vzoru Unit of Work musí být ošetřeno vkládání entit do jednotlivých kolekcí tak, aby mezi těmito kolekcemi nedocházelo k chybnému uložení entit. Pokud je například vkládána entita do kolekce nových objektů, je potřeba ověřit zdali tato entita již není v kolekci upravených nebo smazaných entit. Pokud by entita byla již v kolekci upravených nebo smazaných entit, musela by být předtím, než byla označena pro úpravu nebo smazání, vytvořena a nemá smysl ji vytvářet znovu.

Každá třída rozšiřující třídu `UnitOfWork` musí udržovat kolekce s novými, upravenými a smazanými entitami i po vypnutí aplikace. Instance těchto tříd spolu s těmito kolekcemi jsou však za běhu aplikace uložené v operační paměti zařízení, která je po ukončení aplikace systémem smazána, a tím by byly ztraceny všechny změny, které byly provedeny na entitách. Je proto potřeba ukládat tyto kolekce do perzistentního úložiště zařízení.

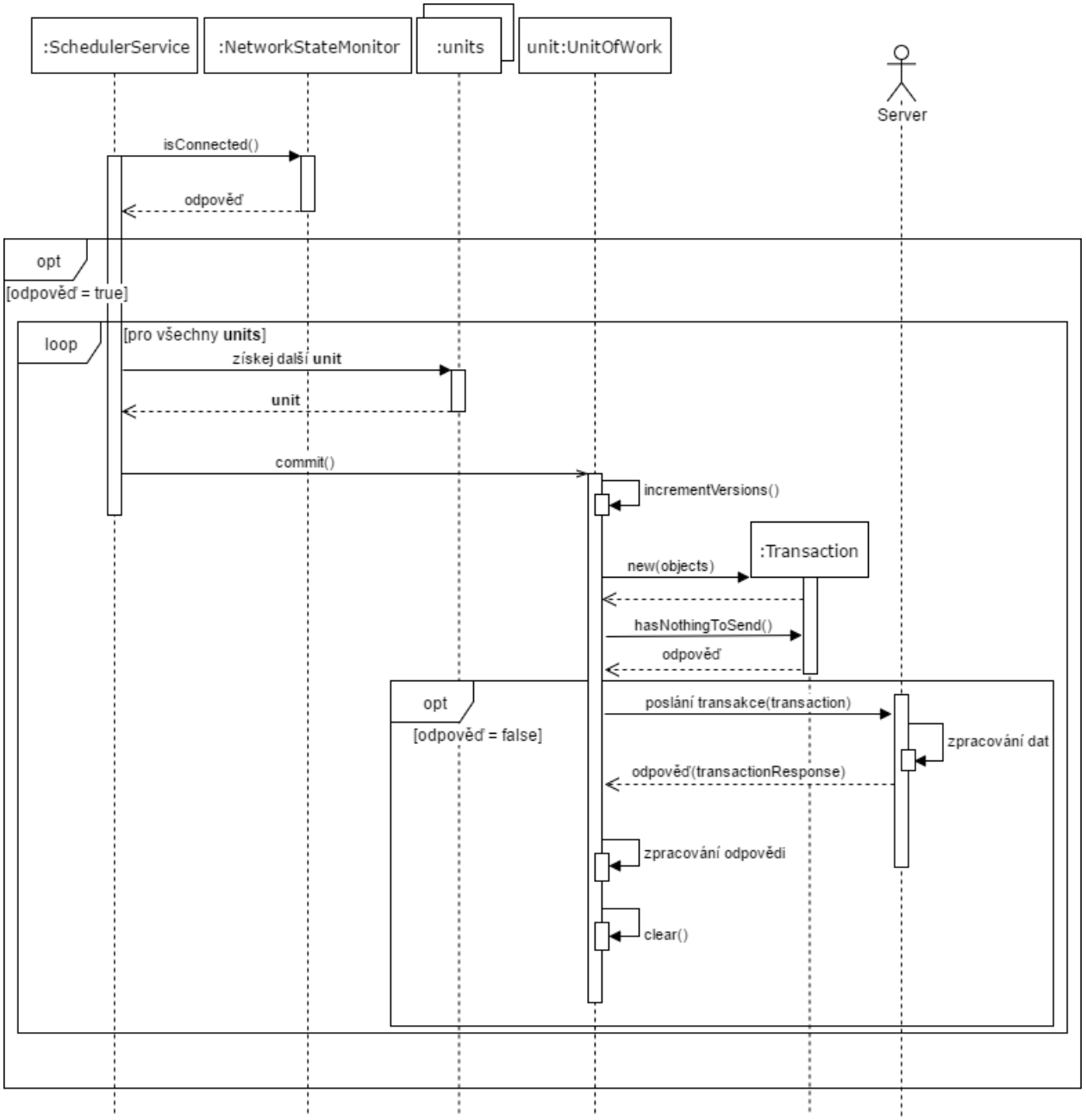
V knihovně je tento problém řešen tak, že při každém přidání entity do jedné z kolekcí je daná kolekce serializována do souboru v interním úložišti zařízení. Každá kolekce v dané třídě má svůj vlastní soubor, do kterého se serializuje a ze kterého se po spuštění aplikace načítá, aby nedocházelo k souběžným problémům při čtení a zápisu souboru. Kolekce s entitami jsou serializovány do souboru v binárním formátu.

V navrhované knihovně je při každém vložení nové entity do kolekce celý soubor pro danou kolekci smazán a poté je vytvořen nový. Tento přístup není ideální, protože zbytečně maže celý soubor místo toho, aby do souboru pouze připsal přidaná data. Lepším řešením by bylo použít pro serializaci entit soubor ve formátu XML, ve kterém by byla každá entita uložena pod danou značkou. Při přidání nové entity by pak stačilo jen přidat na konec souboru novou značku reprezentující tuto novou entitu.

Při spuštění aplikace je potřeba tyto serializované kolekce nahrát zpátky do paměti. Pro tento účel slouží metoda `restoreStoredState`, která je volána v konstruktoru třídy `UnitOfWork`. Po poslání dat na server jsou všechny soubory serializovaných kolekcí smazány, aby při dalším spuštění aplikace nebyla načtena data, která již nejsou pro klienta relevantní. Názvy uložených souborů jsou ve formátu `<název třídy>_<identifikátor kolekce>.ser`.

3.6.6 Posílání a stahování dat

Klient průběžně posílá změny provedené uživatelem na server a stahuje si z něj nová data. Posílání změn na server je znázorněno v sekvenčním diagramu na obrázku 8.



Obrázek 8: Sekvenční diagram - posílání dat na server

V navrhované knihovně má na starost posílání a stahování dat třída `SchedulerService`. Tato třída v pravidelných intervalech posílá změny provedené na klientské části systému na server a poté ze serveru stahuje nejnovější data. V ukázkové implementaci se interval pro tuto třídu nastavuje v souboru *settings.xml* pod názvem *scheduler_period* (soubor `settings.xml` se nachází ve složce `DSSclient/app/src/main/res/values`). Při startu aplikace se musí do instance třídy `SchedulerService` zaregistrovat všechny třídy, které rozšiřují abstraktní třídu `UnitOfWork` pro entity, u kterých je potřeba, aby se posílala a stahovala jejich data. Poté musí být`SchedulerService` spuštěn pomocí metody *start*. Při ukončení programu by měl být `SchedulerService` ukončen metodou *stop*.

Před posíláním změn a stažením dat ze serveru si však nejdříve třída `SchedulerService` pomocí síťového monitoru zjistí, zda je klient připojen k síti (metoda `isConnected`). Síťový monitor je komponenta, která monitoruje stav připojení k síti a poskytuje informace o stavu připojení ostatním komponentám. Funkčnost této komponenty může být závislá na vybrané platformě, a proto knihovna poskytuje pouze rozhraní této komponenty (rozhraní `NetworkStateMonitor`) a nechává na uživateli knihovny, aby ji implementoval. Ve vzorové implementaci je tato komponenta implementována ve třídě `AndroidNetworkStateMonitor`, která je specifická pro platformu Android.

V případě, že klient k síti připojen není, neprovede `SchedulerService` žádnou operaci. Pokud připojen je, tak pro každou rozšiřující třídu `UnitOfWork`, která byla zaregistrována při startu aplikace, potvrdí poslání změn na server pomocí metody `commit`. Každá tato třída nejdříve inkrementuje číslo verze všech upravených objektů a poté vytvoří novou transakci, do které v konstruktoru vloží všechny vytvořené, změněné a smazané objekty. Pokud vytvořená transakce neobsahuje žádné objekty, neodešle se. Pokud prázdná není, odešle se transakce na server, kde je zpracována. Když je odpověď serveru zpracována, všechny změny v dané třídě jsou smazány, aby nedocházelo k opětovnému zasílání stejných informací na server.

Pro stahování dat a uchovávání informace o hlavní entitě je v knihovně vytvořeno rozhraní `DataHolder`. Ve vzorové implementaci toto rozhraní implementuje třída `DataHolderImpl`.
Po poslání všech dat na server třída `SchedulerService` zavolá metodu `downloadData`, která je definována v rozhraní `DataHolder`. Tato metoda zašle na server požadavek na získání dat. V tomto požadavku je pak zaslán identifikátor hlavní entity, pro kterou se mají data stahovat, a také časové razítko reprezentující poslední stažení dat daného klienta. Server požadavek zpracuje a zašle klientovi data, která byla od posledního stažení změněna (toto bylo popsáno v kapitole **3.5.4 Snížení zatížení sítě**). Klient projde stažená data a entity, které mají nastavený příznak `toBeUpdated` na hodnotu `true`, uloží nebo aktualizuje do lokální databáze. Entity jsou aktualizovány na základě globálního identifikátoru, protože server nemá ve své databázi uložené lokální identifikátory entit a nemůže je tedy ani klientovi poslat (důvod použití lokálních identifikátorů je popsán v kapitole **3.6.4 Lokální identifikátor**). Pro aktualizaci entit do lokální databáze je použita pomocná třída pro stavbu dotazu z knihovny ORM-Lite.

Zprvu měla být komponenta pro stahování a posíláni dat implementována pomocí abstraktní třídy `JobScheduler`, která je k dispozici v platformě Android. Použitím této třídy by se však stala komponenta závislá na platformě Android, a proto byla nakonec implementována pomocí třídy `ScheduledExecutorService`, jež je součástí Javy. Díky tomu může být tato komponenta použita i jinde než na platformě Android. Příklad použití `ScheduledExecutorService` je zobrazen ve výpise 7. V tomto příkladě je vytvořena instance třídy `ScheduledExecutorService`, která může využít až pět vláken. Metoda *doWork* reprezentuje práci, která se má vykonávat. Tato metoda bude spuštěna každých deset sekund (tento čas je určen druhým parametrem metody *scheduleAtFixedRate*), kde je první spuštění odloženo o pět sekund (první parametr metody).

```
ScheduledExecutorService executorService = Executors.newScheduledThreadPool(5);
executorService.scheduleAtFixedRate(() -> {
    doWork();
}, 5, 10, TimeUnit.SECONDS);
```

Výpis 7: Příklad použití třídy ScheduledExecutorService

3.6.7 Java REST klient

Pro posílání a přijímání dat je v klientské části systému použita knihovna Retrofit[27]. Jedná se o knihovnu, která poskytuje programátorovi funkce HTTP klienta a je dostupná pro platformu Java. Tato knihovna byla zvolena hlavně z důvodů jednoduché použitelnosti, typové bezpečnosti a dobrého výkonu. Oproti jiným knihovnám, které poskytují stejnou funkcionalitu (například Volley[34]), je Retrofit jednodušší na použití a je také aktivněji vyvíjen. Je proto často oblíbenou volbou v komunitě vývojářů[4].

Pro její použití této knihovny nejdříve musí být definováno rozhraní, ve kterém jsou specifikovány metody a jejich parametry, které budou volány. Příklad tohoto rozhraní je ve výpise 8. Pro použití definovaného rozhraní musí být vytvořena instance objektu `Retrofit`. Při tvorbě tohoto objektu jsou nastaveny jeho parametry, jako například URL adresa, na kterou se mají požadavky posílat, nebo objekt zodpovědný za serializaci a deserializaci požadavků na server a jeho odpovědi. Pomocí tohoto objektu poté lze získat implementaci definovaného rozhraní, kterou potom lze používat pro posílání požadavků. Popis parametrů rohraní a jejich použití lze nalézt na oficiální stránce knihovny Retrofit[5]. Ve vzorové implementaci je URL adresa pro Retrofit načítána ze souboru `values.xml`, kde se nachází pod klíčem *server_address* (soubor `values.xml` se nachází ve složce `DSSclient/app/src/main/res/values`). Pro serializaci a deserializaci objektů je stejně jako u serverové části použita knihovna Jackson[22].

[4] https://stackoverflow.com/questions/8267928/android-rest-client-sample
[5] http://square.github.io/retrofit/

```
public interface UserService {

    @Headers("Content-Type: application/json")
    @POST("/users")
    public Call<User> createUser(@Body User user);

    @GET("/users/{userId}")
    public Call<User> getUser(@Path("userId") long id);
}
```

Výpis 8: Ukázka rozhraní pro knihovnu Retrofit

Knihovna Retrofit umí zasílat požadavky synchronním i asynchronním způsobem. Při použití asynchronního režimu knihovna automaticky vytvoří nové vlákno a v něm provede přenos dat. Po získání odpovědi serveru se zavolá předem definovaná obslužná metoda, která odpověď zpracuje.

Při používání této knihovny nastal problém se serializací a deserializací časových razítek. Přestože byl objektu zodpovědnému za serializaci ostatních objektů nastaven formát "den.měsíc.rok hodina:minuty:sekundy", byla na server časová razítka posílána v naprosto jiném formátu. Časová razítka s tímto formátem nemohl server rozpoznat, a proto všechny požadavky s časovými razítky nebyly obslouženy. Po prozkoumání zdrojových kódů knihovny Retrofit bylo zjištěno že, objekty, které jsou nastaveny pro serializaci dat, jsou použity pouze pro serializaci těla požadavku (třída zodpovědná za serializaci a deserializaci objektů obsahovala pouze metody `responseBodyConverter` a `requestBodyConverter`). Časová razítka, která byla posílána s požadavky na server, však byla posílána v hlavičce požadavku, a proto nebyla serializovaná do požadovaného formátu.

Tento problém by mohl být vyřešen v případě, že by byla časová razítka posílána v těle dotazu a ne v jeho hlavičce. Požadavek s časovým razítkem je ale na server zasílán s HTTP metodou GET. Podle specifikace HTTP protokolu, pokud metoda HTTP požadavku neobsahuje definovanou sémantiku pro tělo požadavku, by pak by mělo být toto tělo požadavku ignorováno během zpracovávání požadavku serverem[6]. Ve specifikaci metody GET je pak uvedeno, že tato metoda má vracet informaci, která je identifikována pomocí URI v požadavku[7]. To znamená, že tělo zprávy není součástí identifikace zdroje požadavku s HTTP metodou GET. Přestože se jedná pouze o doporučení, není vhodné tento přístup aplikovat, protože by se mohlo stát, že se server bude řídit tímto doporučením a bude ignorovat tělo HTTP požadavků s metodou GET. K tématu posílání požadavků pomocí metody GET s tělem požadavku se vyjádřil i autor

[6] https://tools.ietf.org/html/rfc2616#section-4.3
[7] https://tools.ietf.org/html/rfc2616#section-9.3

REST architektury Roy Fielding[8]. Ve svém komentáři uvádí, že je sice možné posílat požadavek s metodou GET, který má i tělo, ale nikdy to není užitečné.

Lepším řešením je nepřenechávat serializaci časových razítek knihovně Retrofit, ale před každým požadavkem na server serializovat časová razítka na řetězec pomocí třídy `SimpleDateFormat`, která je součásti standardních knihoven Javy. Pro převedení časového razítka na řetězec pomocí této třídy musí být nejdříve vytvořena její instance, kde je v konstruktoru definován formát, se kterým má objekt pracovat. Poté můžeme časová razítka převádět pomocí funkce `format`, která vrací textový řetězec v požadovaném formátu.

3.6.8 Posílání dat v Androidu

V Androidu nemohou být od verze 3.0 (Honeycomb) prováděny síťové operace na hlavním vlákně aplikace [38]. Při pokusu o provádění síťových operací nastane chyba a systém vyhodí výjimku `android.os.NetworkOnMainThreadException`. Používání síťových volání bylo v Androidu zakázáno z důvodu zlepšení odezvy a snížení problémů se zamrzáním aplikací. Kvůli tomuto omezení musí být data v klientovi posílána na jiném než hlavním vlákně. Proto pro provedení synchronního volání musí být vytvořeno nové vlákno a volání se musí provést v něm.

Pro tento účel lze v Androidu použít třídu `AsyncTask`. Tato třída slouží pro provádění krátkých operací mimo hlavní vlákno. Pro jeho využití musí být vytvořena třída, která rozšiřuje třídu `AsyncTask`. Ta musí implementovat metodu `doInBackground`, ve které specifikuje funkcionalitu, kterou je třeba provést mimo hlavní vlákno. Instance této třídy je pak spuštěna pomocí funkce `execute`, které je v parametru předáno pole vstupních argumentů. Pro získání výsledku asynchronní operace musí být na instanci zavolána metoda `get`. Pokud při volání této metody nebyla ještě asynchronní operace dokončena, program počká na její dokončení.

3.6.9 Řešení kolizí verzí entit

Pokud je na server poslán požadavek pro úpravu entity a server zjistí, že entita, kterou chce klient upravit, je zastaralá (její verze je nižší nebo stejná jako verze objektu uloženého v databázi serveru), označí v odpovědi tuto operaci za chybnou a pošle klientovi entitu s nejnovější verzí. Jestliže klientská část systému při procházení výsledků operací úprav entit, jež byly zaslány na server, najde operaci, která selhala kvůli zastaralé entitě, zaregistruje tuto událost třídě implementující rozhraní `ConflictResolver`. Ve vzorové implementaci je pro tento účel vytvořena třída `ConflictResolverImpl`. Uživatel systému je na tuto registraci upozorněn notifikací.

[8] https://groups.yahoo.com/neo/groups/rest-discuss/conversations/messages/9962

Po registraci konfliktu může uživatel daný konflikt vyřešit dvěma způsoby:

- **Vynucení změny** - Pokud uživatel vybere tuto volbu, tak bude zastaralé entitě nastavena její nejnovější verze a bude poslána na server, kde se aplikují změny této entity.
- **Zahození změny** - Při zvolení této volby budou zahozeny všechny změny zastaralé entity.

Zaznamenané konflikty by měly zůstat uloženy v systému i po vypnutí aplikace. Proto, stejně jako u ukládání změn entit, jsou kolekce s konflikty verzí ukládány do souboru, který je ve formátu: `<název třídy>_conflict.ser`. Při opětovném spuštění aplikace jsou tyto kolekce obnoveny z uložených souborů.

Toto řešení konfliktů verzí není ideální, protože uživatel nemá možnost upravit zastaralou entitu před jejím opětovným posláním na server. Navíc by měl být uživatel upozorněn, pokud jsou data, která si zrovna prohlíží nebo upravuje, změněna stažením nových dat ze serveru. Tento pokročilejší přístup k řešení kolizí verzí entit na klientské části systému nebyl implementován z časových důvodů.

Tato funkcionalita by však mohla být implementována použitím návrhového vzoru Observer[39]. Tento návrhový vzor je používán v situacích, kdy je potřeba, aby určité objekty dostávaly informaci o změně stavu jiného objektu. V kontextu tohoto systému by byl návrhový vzor Observer použit tak, že pokud začne uživatel upravovat některou z entit, bude tato entita registrována ke sledování změn. Pokud během úprav entity uživatelem dojde ke stažení dat ze serveru, kde mezi staženými daty bude i nová verze upravované entity, bude část systémů, zodpovědná za úpravu entit, upozorněna na danou změnu. Poté tato část systému upozorní uživatele, který se může rozhodnout, zda bude chtít nadále pracovat s rozpracovanými změnami, nebo jestli chce své změny zahodit a načíst novou verzi entity.

3.6.10 Mazání zastaralých entit

Při synchronizaci dat mezi klientskou a serverovou částí systému je potřeba zajistit, aby entity, které byly smazány na straně serveru, byly taktéž smazány na straně klienta. Server však nemůže uchovávat seznam entit, které byly smazány, protože by byl časem úplně zaplněn. Navíc by ani nebylo jasné, kdy může informaci o smazání entity přestat uchovávat, protože server nemá informace o tom, jestli má ještě některý z klientů tuto entitu uloženou.

Při mazání však lze využít toho, že je na klientskou část systému zasílán celý graf entit, relevantních pro daného klienta. Identifikátory z tohoto grafu entit můžeme porovnat s lokální databází. U entit, které se nacházejí v lokální databázi, ale ne v entitním grafu, se může předpokládat, že byli na serveru smazány a mohou být tedy vymazány i z lokální databáze. Ve vzorové implementaci má mazání zastaralých entit na starost třída `EntityCleaner`. Do této třídy se ukládají identifikátory entit ze staženého grafu, a to během synchronizace stažených dat ze serveru do lokální databáze. Na konci synchronizace je na této třídě zavolána metoda *cleanAll*, která smaže všechny zastaralé entity. Pro mazání využívá třídu pro stavbu dotazů

z knihovny ORMLite. Ve výpise 9 je znázorněna ukázka použití třídy pro stavbu dotazu, který slouží k mazání entit, jež nejsou v dané kolekci. Kolekce identifikátorů entit, které nemají být smazány, jsou v ukázce uloženy v listu *actorIds*. Prvním parametrem v metodě `in` je název sloupce, podle kterého se mají entity identifikovat.

```
List<Long> actorsIds = ....
DeleteBuilder<Actor, Long> deleteBuilder = databaseHelper.getActorsDAO().
    deleteBuilder();
try {
    deleteBuilder.where().not().in("actor_id", actorsIds);
    deleteBuilder.delete();
} catch (SQLException e) {
    e.printStackTrace();
}
```

Výpis 9: Příklad použití třídy pro stavbu dotazu

3.6.11 Konfigurační management klientské části

Sestavení projektu a konfigurace prostředí

Platforma Android pro sestavení a správu aplikace používá nástroj Gradle[28]. Tento nástroj je podobný Mavenu, který je používán pro sestavování serverové části. Android aplikace se skládají z jednoho nebo více modulů. Každý modul má svůj vlastní konfigurační soubor `build.gradle`, který obsahuje nastavení pro sestavení projektu a popisuje závislosti na externích knihovnách. Tento soubor je i v kořenové složce celého projektu a v tomto souboru je uloženo nastavení společné pro všechny moduly. Gradle používá vlastní doménově specifický jazyk pro soubory `build.gradle`, který je však založený na jazyce Groovy[29]. Gradle umí používat knihovny uložené v repozitáři Mavenu. Stejně jako Maven si Gradle při sestavování projektu nejdříve stáhne všechny potřebné externí knihovny. Gradle je také ovládán pomocí příkazů z příkazové řádky.

Klientská aplikace obsahuje v kořenovém adresáři Gradle Wrapper[30]. Jedná se o skript, který stáhne definovanou verzi Gradlu a poté pomocí něj začne provádět definovaný cíl. Tento skript lze vygenerovat spuštěním příkazu `gradle wrapper`. Díky tomuto skriptu není nutné mít na počítači, kde je potřeba projekt sestavit, nainstalovaný nástroj Gradle. Tento skript lze spustit jak na linuxových distribucích (soubor `gradlew`), tak na operačním systému Windows (soubor `gradlew.bat`). Použitím tohoto skriptu odpadne závislost na konkrétním vývojovém prostředí a částečně i na Gradlu samotném (ve smyslu že nemusí být ručně instalován).

Před sestavením klientské části systému je potřeba nastavit přístupové údaje do repozitáře, kde je knihovna uložena, nebo ji nainstalovat do lokálního Maven repozitáře. Postup jak toho docílit je již popsán v kapitole **3.5.6 Konfigurační management serverové části**. U klientské části je postup stejný, pouze je při instalaci knihovny do lokálního repozitáře potřeba sestavit a nainstalovat místo serverové části knihovny část klientskou (DSSlib-client).

Pro sestavení klientské části vzorového projektu stačí zadat příkaz `gradlew assembleDebug`. Gradle Wrapper stáhne definovanou verzi Gradlu (pokud předtím nebyla stažena) a stáhne externí knihovny, zkompiluje projekt a zabalí ho do souboru `app-debug.apk`, který bude Gradlem vygenerován ve složce `DSSclient/app/build/outputs/apk/debug`. Takto vygenerovaný soubor je určen pro účely vývoje. Pokud je potřeba vygenerovat verzi určenou pro ostré nasazení, musí být nejdříve aplikace podepsána soukromým klíčem a poté sestavena pomocí příkazu `gradlew assembleRelease`. Postup podepsání aplikace soukromým klíčem je popsán na stránkách určené vývojářům androidu[31].

Nastavení projektu se nachází v souboru `settings.xml`, ve složce values , která se nachází v kořenové složce zdrojů projektu (`DSSclient/app/src/main/res/values`). V tomto souboru lze nastavit jméno souboru pro databázi, verzi databáze, adresu serverové části systému a interval pro stažení a posílání dat.

Nasazení

Klientská část systému může být nasazena na zařízení s operačním systémem Android ve verzi 6.0 a vyšší. Pokud je zařízení, na které má být aplikace nasazena, připojeno k počítači, je možné při sestavování klientské části systému použít místo příkazu `gradlew assembleDebug` příkaz `gradlew installDebug`, který aplikaci nejen sestaví, ale i rovnou nasadí na zařízení. Předtím ale musí být v zařízení povoleno ladění přes USB.

Pro nasazení již sestavené aplikace lze použít nástroj Android Debug Bridge (ADB). ADB je všestranný nástroj pro komunikaci se zařízeními s operačním systémem Android . Pro nasazení aplikace pomocí tohoto nástroje je potřeba ve složce se souborem `app-debug.apk` zadat příkaz: `adb -d install app-debug.apk`. I pro tento přístup je potřeba mít v nastavení zařízení zapnuto ladění přes USB.

Pokud aplikaci nelze nainstalovat na zařízení z počítače, může být nasazena i přímo ze zařízení. Nejdříve se zkopíruje soubor `app-debug.apk` na zařízení. Pro tento způsob nasazení je potřeba nainstalovat některý ze správce souborů, protože ve většině zařízení Android není v továrním stavu tento správce souborů nainstalován. Před instalací je také potřeba v nastavení povolit instalování aplikací z neznámých zdrojů. Poté je potřeba ve správci souborů najít soubor `app-debug.apk`, otevřít ho, a potvrdit instalaci aplikace.

4 Testování

Implementovaný systém a knihovnu je vhodné testovat, aby mohla být ověřena jejich kvalita, a z důvodu snížení pravděpodobnosti selhání systému nebo knihovny během jejich používání. Pro testování navržené knihovny jsou vytvořeny testy k testování jednotlivých tříd. Pro serverovou část vzorové implementaci byly vytvořeny testy webového rozhraní a výkonu. Testy mohou ukázat přítomnost vad v softwaru, nikdy však nelze otestovat software stoprocentně, a proto nemohou prokázat jejich absenci. Proto je vhodné při vývoji nespoléhat pouze na tyto testy a testovat systém i manuálně.

4.1 Testování komponent

Pro základní ověření správného fungování knihovny byly vytvořeny automatizované testy tříd (unit testy). Tyto testy ověřují základní funkcionalitu jednotlivých tříd. Každý test třídy by měl být popsán a tento popis by měl obsahovat následující položky:

- Jedinečné jméno/titulek
- Jedinečné ID
- Popis testu
- Předpoklady před spuštěním testu
- Jednotlivé kroky testu
- Očekávané výsledky testu

Pro účely testování komponent je použit framework JUnit [32]. JUnit je open source testovací framework, který používá anotace pro identifikaci metod, které specifikují test. Při testování tříd by měla být testována pouze jedna daná třída (pro testování komunikace mezi více třídami najednou slouží integrační testy). Tato třída však může být závislá na ostatních třídách. Proto se při testování tříd nahrazují závislosti na ostatních třídách zástupnými objekty. Těmto objektům je před spuštěním testu nastaveno, jak mají odpovědět, pokud testovaná třída zavolá některou z jejich metod. Pro tvorbu a konfiguraci zástupných objektů je často používána knihovna nebo framework. V knihovně je pro tento účel použit framework Mockito [37]. Testy komponent je pokryto 89 % kódu klientské části knihovny. V serverové části knihovny je testy pokryto 91 % kódu.

4.2 Funkční testování rozhraní

Otestováno je i rozhraní serverové části systému. Tyto testy probíhají tak, že je na server poslán HTTP požadavek, který simuluje běžný požadavek od klienta. Poté je zkontrolováno, jestli serverová část systému vrátila očekávanou odpověď. Před začátkem testování by měla být databáze systému prázdná, aby nedocházelo ke konfliktům mezi daty uloženými v databázi a testovacími daty. Pro funkční testování rozhraní serverové části je použit nástroj Apache Jmeter[33][9]. Apache Jmeter je open source nástroj navržený pro zátěžové a funkční testování a měření výkonu systémů.

Parametry testů jsou nastaveny v testovacím plánu. Jedná se například o URL adresu a port, na kterém je aplikace spuštěna. Při nasazení aplikace na jiný server stačí pouze pozměnit tyto parametry a poté mohou být testy opět spouštěny.

Před spuštěním funkčních testů rozhraní je nejdříve spuštěn krátký test, který ověří, jestli je systém na dané adrese dostupný a je spuštěn.

Při testování jsou na server postupně poslány požadavky pro vytvoření objektu, úpravu staré verze objektu, úpravu nové verze objektu a smazání objektu. U každé této operace je zkontrolována jestli odpověď serveru odpovídá očekávanému výsledku.

4.3 Výkonnostní testování

Výkonnostní testy se používají k opakované analýze výkonu softwarového systému. Jejich účelem je zjistit, jak se testovaný systém chová pod vysokým zatížením. Pro výkonnostní testování systému je také použit nástroj Apache Jmeter.

Výkonnostní testování probíhalo na odděleném stroji, aby nedocházelo k situaci, kdy by testovací nástroj používáním prostředků počítače ovlivňoval výkon testovaného softwaru, a tím by byly zkresleny výsledky výkonnostních testů. Testovaný systém byl proto před výkonnostním testováním nasazen na server školy, jenž byl poskytnut pro účely testování. Stejně jako u funkčního testování rozhraní by měla být databáze systému prázdná, jinak by mohlo dojít ke ztrátě uložených dat.

Hardwarová konfigurace počítače použitého pro výkonnostní testování:

- **Procesor** - 2x Intel Xeon E5-264 (2.50 GHz, 15 MB Cache, 64 bit, 6 jader, 12 vláken)
- **Operační paměť** - 128 GiB DDR3
- **Pevný disk** - více disků a celkové velikosti 1.6 TB

Testovaný systém byl nasazen na virtuálním serveru, který byl spuštěn na tomto počítači. Na virtuálním serveru byl nainstalován operační systém Ubuntu ve verzi 14.04.5. Operační paměť virtuálního serveru byla omezena na 4 GiB.

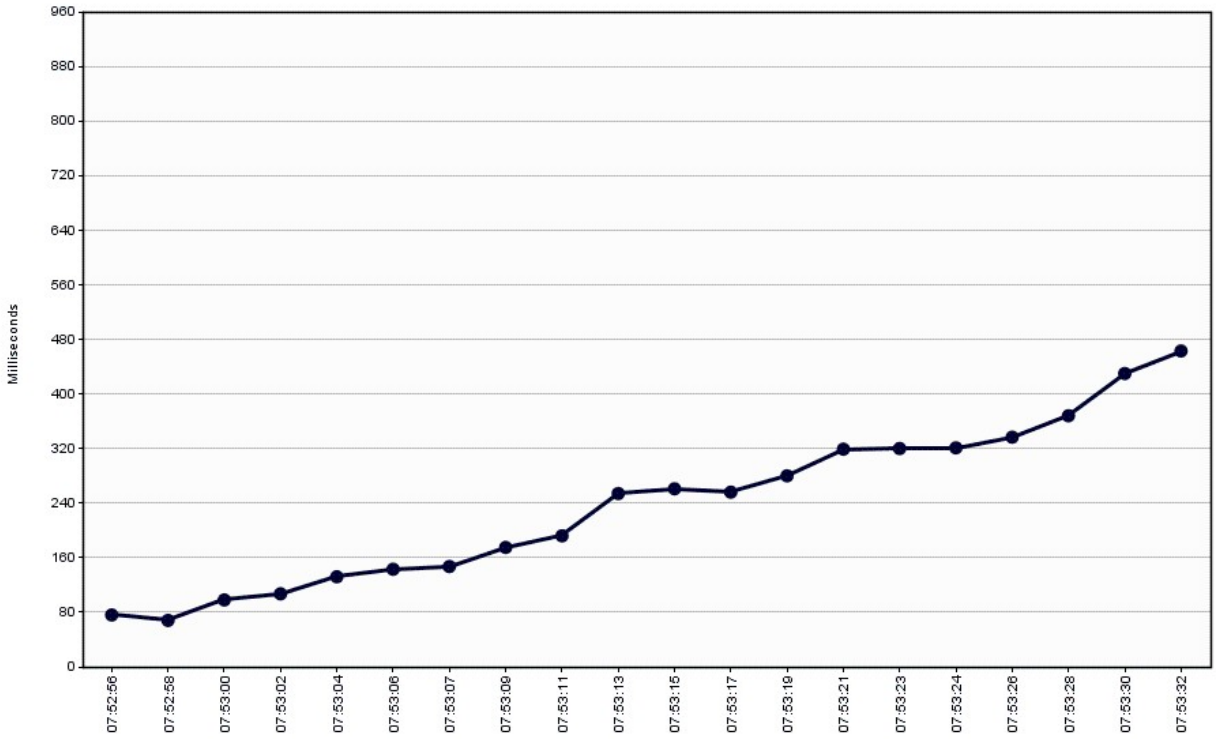
[9] Pro testování byl použit Jmeter ve verzi 4.0

4.3.1 Testování velikostí požadavku

Tento testovací případ analyzuje, jak se systém chová pod zvyšující se velikostí zasílaných požadavků. Během testu jsou na server posílány požadavky, jejichž velikost je postupně zvětšována a je měřeno, jaký vliv má zvětšování velikosti požadavků na čas, který server potřebuje pro zpracování požadavku.

Posílané požadavky obsahují transakce s operacemi nad entitami. Postupně je do transakcí přidáváno více operací a je kontrolován čas, který server potřeboval pro zpracování požadavku. Mezi zasílanými požadavky, je prodleva dvě sekundy, aby měl server dost času odpovědět.

Data posílaných požadavků jsou v souboru `data_pozadavky.csv` pro větší přehlednost testu a snadnější úpravu testovacích dat. Výsledný graf tohoto testu je znázorněn na obrázku 9. Na svislé ose je doba odpovědi serveru a na vodorovné ose je čas průběhu testu.



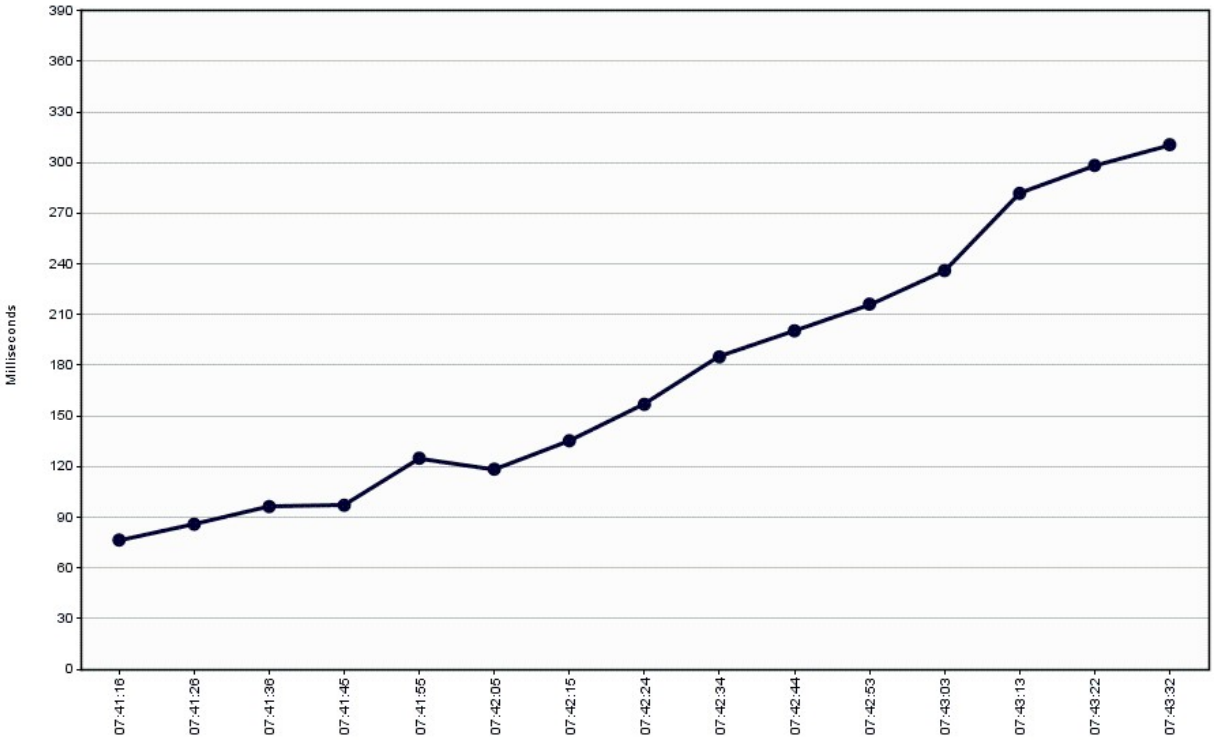
Obrázek 9: Testování velikostí požadavku - graf doby odpovědi serveru

Podle předpokladu se zvyšující se velikostí požadavků stoupá i čas potřebný pro jejich zpracování. Při zaslání transakce, která obsahuje dvacet operací, systém odpoví za zhruba 490 milisekund. Tento výsledek je očekávaný a lze tedy říct, že systém reaguje na zvyšující se velikost požadavků dobře.

4.3.2 Testování počtem uživatelů

V tomto testovacím případě je simulováno běžné používání serverové části systému více uživateli najednou. Během testu jsou na server zasílány požadavky, přičemž je postupně zvyšován počet vláken, která tyto požadavky posílají, čímž je simulován zvětšující se počet uživatelů používajících systém.

Na začátku testu posílá požadavky na server pouze jedno vlákno. Každých deset sekund je spuštěno nové vlákno, které začne posílat požadavky na server až do doby, kdy je dosaženo patnácti současně pracujících vláken. Po dosažení patnácti vláken test ještě deset sekund běží a poté je ukončen. Transakce, které vlákna posílají na server, jsou brány ze souboru `data_uzivatele.csv`. Aby každá transakce vytvářela na serveru stejnou zátěž, mají všechny transakce čtyři operace. Výsledný graf testování je znázorněn na obrázku 10.



Obrázek 10: Testování počtem uživatelů - graf doby odpovědi serveru

Na grafu lze vidět, že se zvyšujícím se počtem uživatelů, kteří používají systém, se zvyšuje i čas potřebný k zpracování jednotlivých požadavků. Když systém dosáhl patnácti souběžných uživatelů, doba jeho odpovědi byla zhruba 310 milisekund. Vzhledem k tomu, že tato doba není příliš vysoká, lze tvrdit, že systém reaguje na zvyšující se počet uživatelů dobře.

5 Závěr

V rámci této diplomové práce byl proveden průzkum problému částečné synchronizace mezi serverovou a klientskou částí systému. Během průzkumu bohužel nebyl nalezen žádný článek nebo software, který by se zabýval přesně danou problematikou.

Poté byl navržen a implementován prototyp knihovny, která pomáhá tento problém řešit. Při návrhu a vývoji knihovny byl kladen důraz na obecnost a jednoduchou konfiguraci knihovny, aby mohla být použita v jakémkoliv typu aplikace, aniž by knihovna musela být jakkoliv modifikována.

Pro ověření správného fungování knihovny byly vytvořeny automatizované testy, které testují správné fungování jednotlivých tříd knihovny. Knihovna byla umístěna do Maven repozitáře[10], odkud může být použita v jakémkoliv projektu používající nástroj Maven nebo Gradle. Je také dostupná v git repozitáři[11].

Dále byl implementován ukázkový systém, který tuto knihovnu využívá k řešení daného problému. Díky použití knihovny je synchronizace dat mezi serverovou a klientskou částí ukázkového systému transparentní. To znamená, že pokaždé, když jsou na klientské části systému získána data z lokální databáze, je zajištěno, že se jedná o nejnovější data, která může mít klient k dispozici, aniž by muselo být explicitně žádáno o synchronizaci dat se serverem.

Pro serverovou část vzorové implementace byly vytvořeny testy, které ověřují správnou funkčnost poskytovaného rozhraní. Byly takové vytvořeny výkonnostní testy, pomocí kterých lze ověřit, jak se daný systém chová pod vysokým zatížením. V rámci této práce byly tyto testy spuštěny a vyhodnoceny. Výsledky výkonnostních testů odpovídaly očekávaným hodnotám (očekávalo se, že čas odezvy serveru bude menší než jedna sekunda). Ukázková implementace systému je dostupná v git repozitáři[12].

[10] Maven repozitář knihovny: https://artifactory.cs.vsb.cz/students-work/com/vsb/
[11] Git repozitář knihovny: https://git.cs.vsb.cz/hob0010/DSSlib
[12] Git repozitář ukázkové implementace systému: https://git.cs.vsb.cz/hob0010/data-storage-synchronizations

Literatura

[1] *Scopus* [online]. Elsevier B.V., 2018 [cit. 2018-01-21]. Dostupné z: https://www.scopus.com/

[2] *Web of Science* [online]. Clarivate analytics, 2018 [cit. 2018-01-21]. Dostupné z: http://apps.webofknowledge.com/

[3] SCHIPER, Nicolas, Rodrigo SCHMIDT a Fernando PEDONE. *Optimistic Algorithms for Partial Database Replication*. Lugano, Switzerland, 2006. University of Lugano.

[4] *Couchbase mobile* [online]. Couchbase [cit. 2018-01-21]. Dostupné z: https://www.couchbase.com/products/mobile

[5] THOMAS FIELDING, Roy. *Architectural Styles and the Design of Network-based Software Architectures*. Oakland, California, U.S., 2000. Disertace. University of California.

[6] TANENBAUM, Andrew S. a Maarten van. STEEN. *Distributed systems: principles and paradigms*. 2nd ed. Upper Saddle RIiver, NJ: Pearson Prentice Hall, c2007. ISBN 0-13-239227-5.

[7] FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, c2003. ISBN 0-321-12742-0.

[8] Java Persistence API [online]. Oracle [cit. 2018-02-22]. Dostupné z: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[9] Hibernate ORM [online]. [cit. 2018-02-22]. Dostupné z: http://hibernate.org/orm/

[10] Extensible Markup Language (XML) 1.0 (Fifth Edition) [online] W3C [cit. 2018-02-26]. Dostupné z: https://www.w3.org/TR/REC-xml/.

[11] Java Reflection API [online]. Oracle [cit. 2018-02-27]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html

[12] XML Parsing for Java [online]. Oracle [cit. 2018-02-27]. Dostupné z: https://docs.oracle.com/cd/B28359_01/appdev.111/b28394/adx_j_parser.htm

[13] TIOBE Index for March 2018 [online]. Toibe [cit. 2018-03-05]. Dostupné z: https://www.tiobe.com/tiobe-index/

[14] GitHut 2.0 [online]. [cit. 2018-3-5]. Dostupné z: https://madnight.github.io/githut/#/pushes/2017/4

[15] GitHub [online]. GitHub Inc., 2018 [cit. 2018-03-05]. Dostupné z: https://github.com/

[16] Spring Framework [online]. Pivotal Software Inc., 2018 [cit. 2018-03-07]. Dostupné z: https://projects.spring.io/spring-framework/

[17] IETF [online]. [cit. 2018-03-08]. Dostupné z: http://ietf.org/

[18] Deprecating Secure Sockets Layer Version 3.0 [online]. IETF, 2015 [cit. 2018-03-08]. Dostupné z: https://tools.ietf.org/html/rfc7568

[19] Mobile Operating System Market Share Worldwide [online]. StatCounter [cit. 2018-03-08]. Dostupné z: http://gs.statcounter.com/os-market-share/mobile/worldwide

[20] SQLite [online]. SQLite [cit. 2018-03-09]. Dostupné z: https://www.sqlite.org

[21] Apache Maven [online]. The Apache Software Foundation, 2018 [cit. 2018-03-12]. Dostupné z: https://maven.apache.org/

[22] Jackson Project Home [online]. [cit. 2018-03-12]. Dostupné z: https://github.com/FasterXML/jackson/

[23] Jackson-datatype-hibernate [online]. [cit. 2018-03-12]. Dostupné z: https://github.com/FasterXML/jackson-datatype-hibernate

[24] RICE, David. Optimistic Offline Lock [online]. [cit. 2018-03-12]. Dostupné z: https://martinfowler.com/eaaCatalog/OptimisticSketch.gif

[25] MARTIN, Fowler.Data Mapper [online]. [cit. 2018-03-14]. Dostupné z: https://martinfowler.com/eaaCatalog/dataMapper.html

[26] Apache Tomcat [online]. Apache Software Foundation, 2018 [cit. 2018-03-14]. Dostupné z: http://tomcat.apache.org/

[27] Retrofit [online]. Square Inc. [cit. 2018-03-15]. Dostupné z: http://square.github.io/retrofit/

[28] Gradle [online]. Gradle Inc. 2018 [cit. 2018-03-15]. Dostupné z: https://gradle.org/

[29] Groovy [onilne]. Apache Groovy project, 2018 [cit. 2018-03-15]. Dostupné z: http://groovy-lang.org/

[30] The Gradle Wrapper [online]. Gradle Inc. 2018 [cit. 2018-03-19]. Dostupné z: https://docs.gradle.org/current/userguide/gradle_wrapper.html

[31] Sign Your App [online]. [cit. 2018-03-19]. Dostupné z: https://developer.android.com/studio/publish/app-signing.html

[32] JUnit [online]. The JUnit Team, 2018 [cit. 2018-03-20] Dostupné z: https://junit.org/

[33] Apache JMeter [online]. Apache Software Foundation [cit. 2018-03-20]. Dostupné z: https://jmeter.apache.org/

[34] Volley [online]. [cit. 2018-03-21]. Dostupné z: https://github.com/google/volley

[35] The Transport Layer Security (TLS) Protocol [online]. IETF, 2006 [cit. 2018-03-22] Dostupné z: https://tools.ietf.org/html/rfc4346#section-1

[36] OAuth 2.0 [online]. [cit. 2018-03-22]. Dostupné z: https://oauth.net/

[37] Mocktio [online]. [cit. 2018-04-04]. Dostupné z: http://site.mockito.org/

[38] NetworkOnMainThreadException [online]. [cit. 2018-09-04]. Dostupné z: https://developer.android.com/reference/android/os/NetworkOnMainThreadException.html

[39] Observer Design Pattern [online]. [cit. 2018-11-04]. Dostupné z: https://sourcemaking.com/design_patterns/observer

[40] Jan Kožusznik. Mobile Device Synchronisation with Central Database based on Data Relevance. *arXiv:1804.03889 [cs]*, April 2018. arXiv: 1804.03889.

[41] When you start talking about numbers as small as 2^{-122}, you have to start looking more closely at the things you thought were zero [online]. [cit. 2018-17-04]. Dostupné z: https://blogs.msdn.microsoft.com/oldnewthing/20160114-00/?p=92851

A Seznam příloh na CD/DVD

- `HOB0010.pdf` ..... Tento text ve formátu PDF/A
- `data-storage-synchronizations` ..... Git repozitář se vzorovou implementací systému
  - `DSSclient` ..... Klientská část vzorové implementace
  - `DSSserver` ..... Serverová část vzorové implementace
- `DSSlib` ..... Git repozitář s prototypem knihovnou
  - `DSSlib-core` ..... Základní část knihovny
  - `DSSlib-server` ..... Serverová část knihovny
  - `DSSlib-client` ..... Klientská část knihovny
- `Testy`
  - `Rozhraní` ..... Testy rozhraní serverové části pro JMeter
  - `Výkon` ..... Testy výkonu serverové části pro JMeter